Analysing Java Identifier Names

Thesis

Analysing Java Identifier Names

Simon Butler B.Sc. (Hons) (Open)

A thesis submitted to

The Open University

in partial fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing

Department of Computing and Communications
Faculty of Mathematics, Computing and Technology
The Open University

September 2015

Abstract

Identifier names are the principal means of recording and communicating ideas in source code and are a significant source of information for software developers and maintainers, and the tools that support their work. This research aims to increase understanding of identifier name content types — words, abbreviations, etc. — and phrasal structures — noun phrases, verb phrases, etc. — by improving techniques for the analysis of identifier names. The techniques and knowledge acquired can be applied to improve program comprehension tools that support internal code quality, concept location, traceability and model extraction.

Previous detailed investigations of identifier names have focused on method names, and the content and structure of Java class and reference (field, parameter, and variable) names are less well understood.

I developed improved algorithms to tokenise names, and trained part-of-speech tagger models on identifier names to support the analysis of class and reference names in a corpus of 60 open source Java projects. I confirm that developers structure the majority of names according to identifier naming conventions, and use phrasal structures reported in the literature. I also show that developers use a wider variety of content types and phrasal structures than previously understood. Unusually structured class names are largely project-specific naming conventions, but could indicate design issues. Analysis of phrasal reference names showed that developers most often use the phrasal structures described in the literature and used to support the extraction of information from names, but also choose unexpected phrasal structures, and complex, multi-phrasal, names.

Using `Nominal` — software I created to evaluate adherence to naming conventions — I found developers tend to follow naming conventions, but that adherence to published conventions varies between projects because developers also establish new conventions for the use of typography, content types and phrasal structure to support their work: particularly to distinguish the roles of Java field names.

Acknowledgements

I would first like to thank my supervisors, Dr. Michel Wermelinger, Dr. Yijun Yu and Professor Helen Sharp, without whose willingness to take me on as a student, their support, wise advice, patience and forbearance I would have been unable to complete my research. Secondly, I would like to thank the staff of the Computing and Communications Department at the Open University for their support over the years: questions at departmental conferences, discussions and suggestions have all been very helpful and very much appreciated. In particular I am extremely grateful to Professor Marian Petre for the time she spends supporting the Department's PhD students and organising online sessions for part-time students.

My fellow students, both part- and full-time, have also been an important source of support, advice, friendship and intellectual stimulation. I had the privilege to be part of a relatively large and very enthusiastic cohort of part-time students who tried to make the most of the opportunities available. Thank you in particular to Katie Wilkie, Mike Giddings, Richard Doust, Chris Ireland, Peter Coles, Liz Clarke, Tezcan Dilshener, Tamara Lopez, Rien Sach, Minh Tran, Lionel Montrieux, Paul Warren and Renato Cortinovis.

The Statistics Advisory Service at the Open University, and in particular Dr Álvaro Faria, have been a valuable source of gratefully received advice. I am also grateful to Dr Ray Buse and Dr Westley Weimer of the University of Virginia for kindly allowing me to use their readability metric tool.

And lastly, but most definitely not least, I would like to thank my family for their support. Tracey, my wife, who has tolerated my research and its impact on domestic life, and who is looking forward to a new chapter in our lives. Yarrow, our daughter, who managed to leave home twice during the course of my research — while there was definitely correlation, causation was not so readily demonstrated (I hope). And also my aunt and uncle, Diana and Gerald Downing, without whose timely generosity my research might not have progressed beyond its early stages.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Source code serves two purposes. The first is, through the medium of a compiler or interpreter, to provide a computer with the necessary instructions to execute the program expressed in the source code. The second is to provide a means of communicating the problem solutions used to create the program. Modern software projects consist of large amounts of source code — sometimes many millions of lines of code — often written by multiple software developers, some of whom may never have met, and maintained by other developers who may not have been part of the original development team. The costs of poor communication in source code may be, at best, additional time spent understanding the code prior to undertaking maintenance tasks, and, at worst, bugs that affect performance, and have commercial consequences.

Identifier names are a significant source of information for software developers and maintainers, and the tools that support their work. Source code is read repeatedly by developers, and at many levels of detail: for example, to extract package structure, to identify aspects of software architecture, and to identify the implementation of particular functionality for maintenance. Names can be used to support requirements tracing (Antoniol *et al.*, 2002) and concept location for program maintenance (Dilshener and Wermelinger, 2011; Hill, 2010; Marcus and Poshyvanyk, 2005), and to determine the consistency of concepts expressed in source code (Raţiu, 2009; Falleri *et al.*, 2010; Nonnen *et al.*, 2011).

Identifier names are strings consisting of abbreviations, acronyms and natural language words. At their simplest names are single letter abbreviations, such as `i` used to represent a generic integer value, but may also be compounds of words representing entities (e.g.

`TypeName`), and states (e.g. `isEmpty`), as well as more complicated forms including sentence-like structures. This research investigates two aspects of names: *content type* and *phrasal structure*. By content type I mean the classification of the lexical content of individual tokens: whether tokens are abbreviations, specialised abbreviations familiar to software practitioners, acronyms or words. The term phrasal structure refers to the grammatical forms of names suggested in naming conventions (Gosling *et al.*, 2014; Vermeulen *et al.*, 2000) and the widely used model of names observed by Liblit *et al.* (2006) that names are phrases, or something that might be recognised as part of a phrase.

Extraction of information from identifier names by software engineering tools to support the work of developers is constrained by limitations to the understanding of how developers structure identifier names, and what types of content are included in names. Indeed some proposed methods of extracting information from identifier names assume developers follow naming conventions: standardised rules, proposed by language designers, companies and software practitioners, of the typography and content types used in names to make them more readable. Liblit *et al.* (2006) observed the use of a wider variety of phrasal structures in identifier names than suggested by naming conventions. Abebe and Tonella (2010) and Hill (2010) use Liblit *et al.*'s observations to support the extraction of information from names. Such approaches effectively extract information from a large proportion of names, but not all.

The hypothesis investigated in this dissertation is that *developers use content types, including natural language content, in Java class and reference identifier names in ways that are richer and more varied than those specified in naming conventions, and described in the academic literature.* A consequence of this hypothesis being correct is that software that processes identifier names needs to be capable of intelligently processing common, unusual and rare forms of name and their content in order to provide its intended service to the end-user.

Method names in Java — one of the four most widely used industrial programming languages — have been studied in detail and much is known about their composition and structure (Høst and Østvold, 2008, 2009). However, there have been only limited investigations of class and reference (field, formal argument and local variable) names, which constitute around 62% of unique names and 72% of name declarations in the corpus of 60 FLOSS (*F*ree/*L*ibre and *O*pen *S*ource *S*oftware) Java projects (Chapter 3) studied. Limited understanding of

the content and structure of Java class and reference names constrains the effectiveness of automated techniques developed by software engineering researchers that process and rely on those names as a source of information.

This research seeks to improve understanding of the content and forms of class and reference identifier names created by developers. To that end I seek to answer the research question:

> **"What types of content and phrasal structure do developers use in Java class and reference names?"**

1.1 Approach

The names investigated in this research are taken from software projects written in the Java programming language. Java is strongly typed, which means, for example, where a value is declared boolean it may hold only a boolean value, and, unlike C and C++, numeric values cannot be treated as boolean values. Naming conventions suggest that developers name boolean identifiers in a particular way, and empirical observations confirm this does happen (Liblit *et al.*, 2006). The advantage for name research of strong typing is that the role of a name is defined by its type, and type does not change during execution.

The large number of names in software projects can make extensive analysis time-consuming and, consequently, some empirical studies that analyse names have focused on names found in a few projects. Høst and Østvold (2007) observed that names reflect the cognition and idiosyncrasies of developers. Extending this argument to teams of developers implies that the names found in any given project will reflect the idiosyncrasies of the development team, as well as the application domain. Accordingly, studying the names found in a single software project may not lead to generalisable conclusions, thus greater understanding of names is likely to be achieved through the study of multiple projects. To support the study names from a larger number of projects, without the overhead of repeatedly extracting names from source code, I devised a means of creating and storing a corpus of names drawn from a large number of diverse projects which is described in Chapter 3.

Names have first to be divided into their constituent components or tokens, prior to analysis. Existing tools for the tokenisation of identifier names have limited functionality

and have not been made available by the researchers that created them. To support the work described in this dissertation, I developed a new solution to identifier name tokenisation that addresses some of the limitations of the published solutions (Chapter 4).

There are two principles that guide my approach. The first is that, as far as possible, I avoid the assumption that software developers create only identifier names that adhere to naming conventions, to ensure the content and structure of names is investigated without bias. The second is that each technique developed should be able to be used in a practical tool for software developers, which has implications for the design and publication of any software solutions developed. Any technique developed should be able to be deployed on a developer's workstation and, ideally, be responsive enough to use in a typical IDE (Integrated Development Environment). In addition, the implementation of any technique should be published as a library so that it can be integrated with other tools.

1.2 Dissertation Overview

Chapter 2 describes a pilot study that provides evidence of a link between the quality of identifier names and internal source code quality. The pilot study develops a connection made by Buse and Weimer (2008) between source code readability and software quality. Buse and Weimer's readability metric is based on a number of qualities of source code, but does not incorporate attributes of identifier names other than length. Given the multiple quality attributes that might be assigned to a name, I hypothesised a connection between naming and code quality. The pilot study applies a validated notion of identifier name quality to assess the quality of names, and the relationship between names judged to be poor quality and areas of source code identified as being of lesser quality by both static analysis and source code metrics. The investigation raised a number of questions, including what constitutes a good quality identifier name and how to analyse names to determine their quality, and informs the hypothesis investigated in this dissertation.

Before addressing the main research question, I investigate three research questions which focus on the creation of tools to support the research.

In Chapter 3 I discuss the techniques developed to extract identifier names from source code and store them to support further analysis. In existing source code models identifier names are attributes of abstract syntax tree (AST) nodes, which means names are not directly

accessible. To identify an alternative model to support this research I investigate the research question:

> **RQ 1 How can a model for source code be created where identifier names and named AST nodes are both first class citizens?**

The resulting model and its implementation as a database is used to create a corpus of names and metadata from 60 open source Java projects. The corpus is the source of names studied in the remainder of this dissertation. Names can be recovered from individual or multiple projects and analysed without the overhead of extracting names from source code each time. The metadata collected with the names means that the methods applied to the names in the database can be applied with little modification, if any, to names on an abstract syntax tree (AST). The tools and techniques described in this dissertation can be applied both to develop software engineering tools further and used in an IDE to support the work of developers.

An early step in the analysis of identifier names involves splitting or tokenising the name. When beginning this research there were limitations to the main techniques developed for the tokenisation of names that constrained detailed analysis of names. Methods had been implemented to tokenise identifier names where developers follow simple typographical conventions. However, three problems remained: the tokenisation of names containing an ambiguous typographical feature; 'single case' names, where typographical conventions have been ignored; and the tokenisation of names containing digits, for which there are no typographical conventions. The former issues are related and Chapter 4 addresses the following two research questions:

> **RQ 2 How can more effective mechanisms to tokenise names with ambiguous or no word boundaries be developed?**; and

> **RQ 3 How can effective mechanisms to tokenise names containing digits be developed?**

The techniques developed in Chapter 4 are implemented in the identifier name tokenisation tool (`INTT`), a Java library, that is used to support the remainder of the research.

Having implemented tools to support the analysis of names, the remainder of the dissertation focuses on the principal research question. The research question is decomposed into

6 subsidiary questions (RQ 4–RQ 9) that each focus on an aspect of the principal research question. Chapter 5 focuses on class names examining phrasal structure and the influence of inheritance on names to address RQ 4 and RQ 5. Chapters 6 and 7 examine reference names. RQ 6 and RQ 7 are addressed in Chapter 6 through investigation of the content types and phrasal structure of reference names. RQ 8 and RQ 9, answered in Chapter 7, investigate developers' compliance with naming conventions.

Specifically, Chapter 5 investigates class names by addressing the research questions:

RQ 4 What phrasal structures do developers use in class names?; and

RQ 5 How do developers incorporate super class and interface names in class names?

RQ 4 is investigated by analysing the structure and content of class names using the novel approach of training a part of speech (PoS) tagger model for class names. Analysis to answer RQ 5 investigates patterns of single generation inheritance in the corpus, identifying elements of names from super class and super types that occur in the class name.

Attention turns to reference names in Chapter 6 to find answers to the following research questions:

RQ 6 What content types do developers use to create reference names, and to what extent is each content type used?; and

RQ 7 What phrasal structures do developers use in reference names, and how are they related to Liblit *et al.*'s metaphors?

RQ 6 is answered with a survey of the diversity of name composition in the corpus. Names composed of words and acronyms only are analysed using a specially trained PoS tagger model to answer RQ 7.

So far, content types and phrasal structures have been examined with only limited reference to their context. Naming conventions, such as those specified by Gosling *et al.* (2014) and Vermeulen *et al.* (2000), specify the forms of name that should be used, and the circumstances they should be used in. Conventions offer the developers of tools that process names additional information that may support their analyses. Given the variety of name structures and content types observed in the preceding chapter the question arises whether developers

follow naming conventions, an assumption that researchers have used to support techniques to extract information from identifier names. However there is limited empirical evidence of the extent to which conventions are followed. In Chapter 7 I investigate the adherence to naming conventions of reference names in the corpus by asking the research questions:

> **RQ 8 To what extent do projects adhere to particular naming conventions or style?**; and
>
> **RQ 9 Do some naming conventions tend to be broken more frequently than others?**

Adherence to naming conventions is evaluated using `Nominal`, a Java library I developed to support the research, that evaluates characteristics of identifier names and contains a declarative language to specify naming conventions.

Chapter 8 draws together conclusions about the techniques developed and the findings of this research, and identifies some areas of future work.

The appendices include a glossary of terms, a list of the projects included in the corpus, the database schema used for the corpus, software and data created during the research, and a list of Penn Treebank tags. A road map of the dissertation is given in Figure 1.1 on page 8.

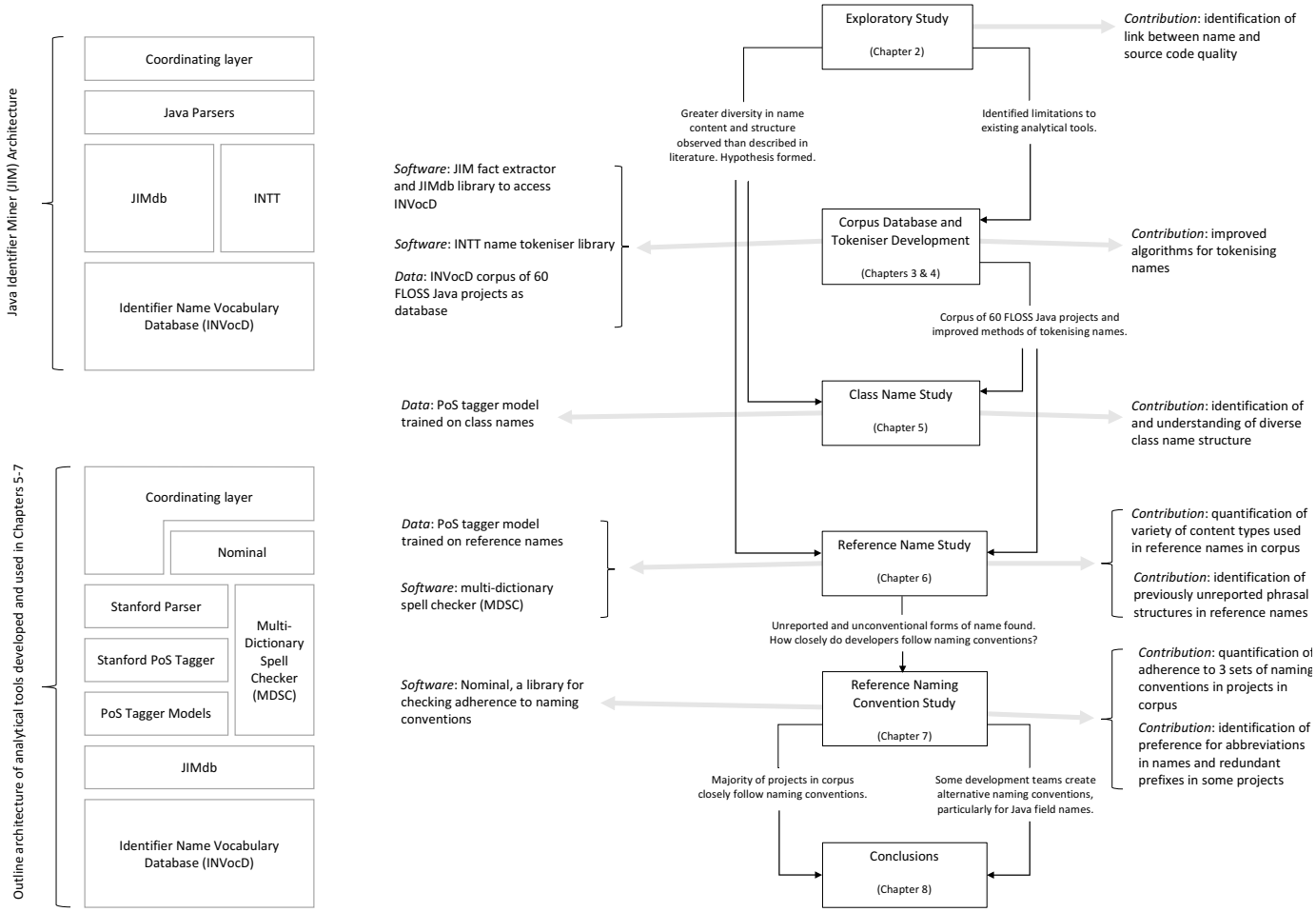


Figure 1.1: Schematic of the road map through dissertation with major contributions and outline architectures of the tools indicating the role played by components created during the research.

1.3 Publications

Parts of this dissertation are based on published conference papers, which are listed below with the chapters they contribute to.

Butler, S.; Wermelinger, M.; Yu, Y. & Sharp, H. (2009) Relating Identifier Naming Flaws and Code Quality: an empirical study. In *Proceedings of the Working Conference on Reverse Engineering*, Lille, France, pp. 31–35. IEEE Computer Society. (Chapter 2)

Butler, S.; Wermelinger, M.; Yu, Y. & Sharp, H. (2010) Exploring the Influence of Identifier Names on Code Quality: an empirical study. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, Madrid, Spain, pp. 159–168. IEEE Computer Society. (Chapter 2)

Butler, S.; Wermelinger, M.; Yu, Y. & Sharp, H. (2011a) Improving the Tokenisation of Identifier Names, In *Proceedings of the 25th European Conference on Object-Oriented Programming*, Lancaster, UK, pp. 130–154. Springer, LNCS 6813. (Chapter 4)

Butler, S.; Wermelinger, M.; Yu, Y. & Sharp, H. (2011b) Mining Java Class Naming Conventions. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, Williamsburg, Virginia, USA, pp. 93–102. IEEE. (Chapter 5)

Butler, S.; Wermelinger, M.; Yu, Y. & Sharp, H. (2013) INVocD: Identifier Name Vocabulary Dataset. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, San Francisco, California, USA, pp. 405–408. IEEE. (Chapter 3)

Butler, S.; Wermelinger, M. & Yu, Y. (2015a) A Survey of the Forms of Java Reference Names. In *Proceedings of the 23rd International Conference on Program Comprehension*, Florence, Italy, pp. 196–206. IEEE. (Chapter 6)

Butler, S.; Wermelinger, M. & Yu, Y. (2015b) Investigating Naming Convention Adherence in Java References. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, Bremen, Germany, pp. 41–50. IEEE. (Chapter 7)

The published papers are available at `http://oro.open.ac.uk/view/person/sjb792.html`

Chapter 2

Exploring the Relationship Between Identifier Names and Code Quality

The impact of low quality identifier names on program comprehension is reasonably well understood (Deißenböck and Pizka, 2006; Lawrie *et al.*, 2007b, 2006), but little is known about the extent to which the quality of identifier names might relate to the quality of source code. A correlation between less readable source code and the occurrence of flaws in the code identified by FindBugs (Ayewah *et al.*, 2007) was identified using a readability metric for source code developed by Buse and Weimer (2008). The readability metric, developed using machine learning, was trained to agree with the judgement of 120 human subjects on code readability. Length is the only property of individual names used as a classifier for the metric, and it is likely that the human subjects would have applied a wider range of criteria to judge names than their length when considering the readability of source code. Given poor quality identifier names are a barrier to program comprehension, and may indicate a lack of understanding of the problem, or the solution articulated in the source code, I hypothesise that poor quality identifier names are indicative of lower quality source code.

The hypothesis is explored by seeking to answer the following research question:

> **RQ A What relationship exists between the occurrence of low quality identifier names and lower quality source code?**

To answer the research question I evaluate the quality of names and source code found in the classes and methods of eight Java projects. The quality of names is determined using a

validated set of naming conventions, and source code quality is evaluated using FindBugs, the cyclomatic complexity metric (McCabe, 1976) and the maintainability index (Welker *et al.*, 1997). In addition, I evaluate the readability of methods using a readability metric (Buse and Weimer, 2008) to verify the link between the readability of source code and FindBugs warnings found by Buse and Weimer (2008). I also explore whether the findings may be applied as a low-cost heuristic to identify potentially problematic regions of source code.

2.1 Related Work

Previous research on identifier naming and source code readability focuses largely on program comprehension, particularly the contribution made by the semantic content of identifier names (Deißenböck and Pizka, 2006; Lawrie *et al.*, 2006; Rajlich and Wilde, 2002). Other related research investigates source code readability (Buse and Weimer, 2008), and begins to explore the relationship between identifier names and software quality (Boogerd and Moonen, 2009; Marcus *et al.*, 2008).

A longitudinal study of identifier names by Lawrie *et al.* (2007b) showed that identifier name quality has improved, in terms of an increase in the proportion of dictionary words used in names, during the preceding thirty years. The same study also found that identifiers in proprietary source code typically contained more domain-specific abbreviations than open source code. However, the study also found that identifiers change little following the initial period of software development. This is confirmed by Antoniol *et al.* (2007) who also argue that programmers may be more reluctant to change identifier names than source code, because of the lack of tool support for managing identifier names. Lawrie *et al.* (2006) describe an empirical study which found identifier names composed of dictionary words were more easily understood than those composed of abbreviations or single letters. An earlier study by Takang *et al.* (1996) found experimental subjects expressed a preference for source code with identifier names composed of dictionary words. The study, however, found no statistically significant difference in the performance of program comprehension tasks undertaken using source code with identifier names composed of abbreviations and code with identifiers composed of dictionary words.

Rajlich and Wilde emphasise the importance of identifiers as the primary source of conceptual information for program comprehension (Rajlich and Wilde, 2002). Deißenböck and

Pizka (2006) developed a formal model for the semantics of identifier names in which each concept is represented by just one name throughout a program. The model excludes the use of homonyms and synonyms, thus reducing the opportunities for confusion. The authors found the model to be an effective tool for resolving difficulties with identifier names found during program development and the resulting source code to be more maintainable.

A study of the morphological and grammatical features of identifier names in C, C++, C# and Java by Liblit *et al.* (2006) found that identifiers are best understood within their working context. Instance variables, for example, are coupled with method names in object-oriented languages, and method names are often conceived with this relationship in mind. Field and variable names have grammatical structures that reflect their independence. The grammatical structure of method names is further differentiated by the need to reflect the action the method performs and whether it has side effects, or takes one or more arguments.

Relf (2004) identified a set of cross-language identifier naming style guidelines from the programming literature, and investigated their acceptance by programmers in an empirical study. Relf implemented the naming style guidelines in a tool to help programmers create good quality identifiers and to refactor existing identifiers (Relf, 2005). Refactoring is the process of revising source code to improve its maintainability without changing its outward behaviour (Fowler *et al.*, 1999). A trigger for refactoring is a 'code smell' or 'bad smell', a relatively simple feature of source code, such as a long method, that may indicate a deeper problem with the implementation or design. Fowler's original list of refactorings includes the renaming of method where the name does not reflect the purpose of the method (Fowler *et al.*, 1999). Abebe *et al.* (2009) developed a system to recognise 'lexicon bad smells' – grammatical and other flaws – in identifiers, thereby identifying a wider range of identifier names for possible refactoring.

Two studies by Boogerd and Moonen (2008, 2009) applied the MISRA-C: 2004 coding standard (MIRA Ltd, 2004) to measure the quality of source code before and after bug fixes during the development of two closed source embedded C applications. They found that while compliance with some of the rules increased as defects were fixed, bug fixes also introduced violations of other rules. In other words, code with fewer defects, and hence of higher quality, is deemed to be of lower quality by some of the coding rules. The authors also found that though they could identify rules with a positive influence on software quality

in each of the two studies, the rules did not have consistent effects, including the four rules related to identifiers common to both studies.

A conceptual cohesion metric developed by Marcus *et al.* (2008) for classes, applies latent semantic indexing (LSI) to the identifier names and comments in methods to establish the level of conceptual cohesion between methods within classes. The metric outperformed existing cohesion metrics as a predictor of defect rates for classes. The technique, however, includes comments in the textual analysis and investigates the consistency of identifier name use through lexical similarity only, without any reference to the quality of the identifier names.

In work classifying the lexicon used in Java method identifiers, Høst and Østvold advance the idea that, because of the effort required to select a good identifier name, identifiers reflect the cognitive processes of programmers and designers (Høst and Østvold, 2007). Consequently, identifiers may then reflect the misunderstandings of the creator of the identifier and misdirect the readers of source code. Further work by Høst and Østvold (2008, 2009) connects the natural language content of method name identifiers with the method implementation, and identify the existence of naming errors that they term 'naming bugs' where the method name might conventionally be expected to describe a particular class of functionality, whereas the method implements something else. In other words the method name is misleading to the point that someone invoking the method would quite probably do so in error.

Buse and Weimer (2008) developed a readability metric for Java derived from measurements of, among others, the number of parentheses and braces, line length, the number of blank lines, and the number, frequency and length of identifiers. Using machine learning, the readability metric was trained to agree with the judgement of human source code readers. Buse and Weimer found a significant statistical relationship between the readability of methods and the presence of defects found by FindBugs (2008) in open source code bases. Although their work makes a link between readability and software quality, their notion of readability, intriguingly, ignores the quality of identifier names.

The existing literature establishes the need for good identifier names to support program comprehension. However, only tentative steps have been taken to investigate a relationship to source code quality. The following investigation aims to provide a step in that direction.

2.2 Identifier Name Quality

There are two broad components of identifier name quality: typography and content. I constrain the measurement of identifier quality to typography and the use of known natural language elements, and do not undertake detailed assessments of semantic content and the use of grammar. Rather than apply an arbitrary set of rules derived from a single set of naming conventions, I used a set of empirically evaluated identifier naming guidelines.

Relf derived a set of twenty-one identifier naming style guidelines for Ada and Java from the programming literature (Relf, 2004). Most of the guidelines, which were evaluated in an empirical study of developers, do not deviate significantly from the Java identifier naming conventions (Gosling *et al.*, 2014; Sun Microsystems, 1999) and as they have been developed in other widely used conventions (Vermeulen *et al.*, 2000).

Relf's identifier naming style guidelines combine typography and a simple approach to natural language, but were not intended to be used as rules to evaluate the quality of identifier names. Accordingly I found it necessary to update some guidelines to define more precisely what was not permitted, and renamed some to reflect the proscriptive sense in which I applied them. Whenever a guideline was modified the original intention of cross-language application was retained insofar as possible.

I implemented a subset of Relf's guidelines as tests. The remaining guidelines were not adopted because either they do not reflect recent changes in Java programming practice, or they are general guidelines of good practice from which it is difficult to derive practical proscriptive rules. For example, Relf defines the Same Words guideline as prohibiting the use of identifiers composed of the same words, but in a different order. Whilst superficially attractive, a rule based on this guideline prohibits clear names for reciprocal operations (e.g. `htmlToXml` and `xmlToHtml`) and pairs of words that create semantically distinct identifiers (e.g. `indexPage` and `pageIndex`). Generally, the implementation of each guideline is apparent from its name, and is described and illustrated in Table 2.1. However, the precise implementation of some guidelines requires further explanation:

Capitalisation Anomaly For identifiers other than constants I test for capitalisation of only the initial letter of acronyms as prescribed by Vermeulen *et al.* (2000), i.e. only the initial letter of a component word is capitalised either at word boundaries, or the beginning

Table 2.1: The Identifier Naming Style Guidelines Applied

Name	Description	Example of flawed identifier(s)
Capitalisation Anomaly	Identifiers should be appropriately capitalised.	`HTMLEditorKit`, `pagecounter`, `fooBAR`
Consecutive Underscores	Consecutive underscores should not be used in identifier names.	`foo__bar`
Enumeration Identifier Declaration Order	Unless there are compelling and obvious reasons otherwise, enumeration constants should be declared in alphabetical order.	`enum Card {ACE, EIGHT, FIVE, FOUR, JACK, KING ...}`
Excessive Words	Identifier names should be composed of no more than four words or abbreviations.	`floatToRawIntBits()`
External Underscores	Identifiers should not have either leading or trailing underscores.	`_foo_`
Long Identifier Name	Identifier names of more than twenty-five characters should be avoided where possible.	`getPolicyQualifiersRejected`
Naming Convention Anomaly	Identifiers should not consist of non-standard mixes of upper and lower case characters.	`FOO_bar`
Non-Dictionary Words	Identifier names should be composed of words found in the dictionary and abbreviations and acronyms that are more commonly used than the unabbreviated form.	`strlen`
Number of Words	Identifiers should be composed of between two and four words.	`ArrayOutOfBoundsException`, `name`
Numeric Identifier Name	Identifiers should not be composed entirely of numeric words or numeric words and numbers.	`FORTY_TWO`
Short Identifier Name	Identifiers should not consist of fewer than eight characters, with the exception of `b`, `c`, `d`, `e`, `g`, `i`, `in`, `inOut`, `j`, `k`, `m`, `n`, `o`, `out`, `s`, `t`, `x`, `y`, `z`	`name`
Type Encoding	Type information should not be encoded in identifier names using Hungarian notation or similar	`iCount`

of the identifier, if appropriate, e.g. `HtmlEditorKit` rather than `HTMLEditorKit`. This is an arbitrary choice and it is known that some development teams prefer the use of upper case acronyms.

Non-Dictionary Words A dictionary word was defined as belonging to the English language, because all the projects investigated are developed in English. I constructed a dictionary consisting of some 117,000 words, including inflections and American and Canadian English spelling variations, using word lists from the SCOWL package up to size 70, the largest lists consisting of words commonly found in published dictionaries (Atkinson, 2004). A further 90 common computing and Java terms, e.g. 'arity', 'hostname', 'symlink', and 'throwable' were added. A separate dictionary of abbreviations was constructed, using the criterion that "the abbreviation is much more widely used than the long form, such as URL or HTML" (Sun Microsystems, 1999).

A concern is that development teams may use project, or domain, specific abbreviations and terms, which are not in my dictionary, yet are well understood by the programmers. To address the issue additional dictionaries were created by the analytical software for each application of unrecognised component words that were used in three, five and ten or more unique identifiers. For example, an unrecognised word or abbreviation used in ten or more unique identifiers may be inferred to be a commonly understood term. The frequencies of three, five and ten are arbitrary, but may be seen as representative of the familiarity the development team might have with a given term. Following the creation of the dictionaries, each identifier was tested again for compliance to the Non-Dictionary Words guideline by using a combination of the main dictionary, the abbreviation dictionary, and each of the dictionaries of project-specific words and abbreviations.

Number of Words Relf's Number of Words guideline was intended to encourage programmers to create identifiers between two and four words long. In applying the guideline as a proscriptive rule both identifiers composed of one word and those composed of five or more words are categorised together, which does not allow the contribution made by the occurrence of either flaw to be determined. The issue is addressed, in part, by the creation of an Excessive Words flaw, defined in Table 2.1, which determines identifiers of five or more words to be flawed.

Short Identifier Name I updated Relf's guideline to include more single letter and short identifiers commonly used in Java (Sun Microsystems, 1999; Gosling *et al.*, 2014; Vermeulen *et al.*, 2000) (see Table 2.1).

2.3 Source Code Quality

This section describes the methods used to evaluate source code quality. The investigation reported in this chapter is motivated by Buse and Weimer's development of a readability metric, which demonstrated correlation between reduced readability and the occurrence of FindBugs warnings Buse and Weimer (2008). I use FindBugs to evaluate the code quality of classes and methods, and two source code metrics to evaluate methods. Buse and Weimer's readability metric was also used to provide assessments of the readability of methods.

2.3.1 FindBugs

FindBugs is a static analysis tool for Java that analyses bytecode for *bug patterns*. The type of defects identified by the bug patterns range from possible dereferences of null pointers, which may halt program execution, to Java specific problems associated with an incomplete understanding of the Java language (Ayewah *et al.*, 2007). The latter class of defects include code constructs likely to increase the maintenance effort and code constructs that may have unintended side-effects. FindBugs was used extensively during two days in May 2009 at Google, and software engineers found some 4,000 significant issues with their Java source code as a result (FindBugs, 2008). While FindBugs reports false positives, as does any static analysis tool, FindBugs' perspective on source code quality is suitable for my needs.

2.3.2 Source Code Quality Metrics

My objective is to measure source code quality in a way that reflects the influence of the programmer on source code and the possible impact on the reader. I used cyclomatic complexity (McCabe, 1976) and the three metric maintainability index (Welker *et al.*, 1997) to measure the quality of Java methods.

Cyclomatic complexity provides a ready assessment of the complexity of a method in terms of the number of possible execution paths. I acknowledge that cyclomatic complexity is a somewhat controversial metric (Munson, 2003), but believe that it provides an indication of source code complexity sufficient for my purposes.

The three metric maintainability index (MI) (Welker *et al.*, 1997) is given by:

$$MI = 171 - 5.2 \times \ln(HV) - 0.23 \times V(G) - 16.2 \times \ln(LOC)$$

where LOC is the number of lines of code, $V(G)$ is the cyclomatic complexity and HV is the Halstead Volume (Halstead, 1977), a source code metric determined by the number of operators and operands used, including identifiers. The Halstead Volume is the product of the Halstead Vocabulary and the logarithm of the Halstead Length. The Halstead Vocabulary is the number of unique operators and unique operands, and the Halstead Length is the sum of the number of operators and operands. By incorporating the Halstead Vocabulary, the MI is influenced by the complexity of a unit of source code in terms of the number of identifiers required to implement a solution.

2.4 Methodology

My investigation is conducted with two different units of analysis: the Java class and the Java method. At the class level, the investigation concerns the possibility of co-occurrence of lower name quality and FindBugs warnings. At the method level, the same relationship is examined, and possible relationships between reduced name quality and additional metrics of source code quality, and readability are considered.

2.4.1 Data Collection

I selected eight established open source Java projects for investigation, including GUI applications, programmers' tools, and libraries. The particular projects were chosen to reduce the potential influence of domain and project-specific factors in the investigation. Table 2.2 shows the version and number of classes and methods analysed for each project.

Table 2.2: Source Code Analysed

Project	Version	Classes	Methods
ANT	1.71	796	9146
Cactus	1.8.0	128	926
FreeMind	0.9.0 Beta 20	404	4883
Hibernate Core	3.3.1	1145	12309
JasperReports	3.1.2	1140	12349
jEdit	4.3 pre16	483	5835
JFreeChart	1.0.11	582	8230
Tomcat	6.0.18	1019	11394

I developed a tool to automate the extraction and analysis of identifiers from Java source

code. Java files were parsed and identifiers analysed on the parse tree to establish adherence to the typographical rules for their context, e.g. method names starting with a lower case character. Identifiers were then extracted and added to a central store, with information about their location, and divided into hard words — their component words and abbreviations — using the conventional Java word boundaries of internal capitalisation and underscores (i.e. conservative tokenisation). Identifiers were then analysed by the tool for conformance to Relf's guidelines in Table 2.1, my own Excessive Words guideline, and the Non-Dictionary Words guideline where the dictionary is extended by a set of commonly used hard words.

Where subject applications were found to contain source code files generated by parser generators, or to incorporate source code from third party libraries, those files were ignored to try to ensure only source code written by the applications' development teams was analysed.

The tool recorded the primitive Halstead metrics for each method by, adapting the standard developed for C by (Munson, 2003) and applying it to Java. McCabe's cyclomatic complexity (V(G)) (McCabe, 1976), was also recorded, and LOC for each method, to compute the maintainability index.

Preliminary analysis showed that FindBugs warnings were reported for a minority of classes. Similarly, many of the identifier flaws were found in a minority of classes. Given the absence of a normal distribution of warnings and flawed identifiers, and that the readability metric is a binary classifier, I decided to treat the presence of flawed names and FindBugs warnings as binary classifiers. To create a binary classifier from the maintainability index I used the threshold of 65, established by empirical study (Welker *et al.*, 1997), to identify methods as 'more-maintainable' and 'less-maintainable'. I also applied the cyclomatic complexity metric as a binary classifier. The popular programming literature often advocates that programmers take steps to keep the cyclomatic complexity of individual methods low. Some texts suggest refactoring should be considered when cyclomatic complexity is six or more, and that the cyclomatic complexity of a method should not exceed ten (McConnell, 2004). It is outside the scope of the investigation to examine the merits of such practices or the justification for the chosen thresholds. However, to create binary classifiers from the cyclomatic complexity metric, I adopted thresholds of six and ten to represent methods of moderate and higher complexity. This provides two binary classifiers distinguishing between methods with low complexity and those with a cyclomatic complexity of six or more, and

between methods with low to moderate complexity and those with a cyclomatic complexity of ten or more.

The readability of source code was evaluated using a readability metric tool developed by Buse and Weimer (2008). The readability metric follows a bimodal distribution and is interpreted as binary classifier that identifies source code as 'more-readable' or 'less-readable'.

The readability metric is used to evaluate the readability of methods to ensure that the metric assessed source code as the human reader would see it. Java source code files contain one or more top-level classes, each of which may contain member classes. Both types of class may contain methods. I recorded as methods, only those contained either by top-level classes or by member classes directly contained by top-level classes. Any local and anonymous classes contained within a methods are recorded as part of the containing method and not separately. For example, if a method contains an anonymous class, the total cyclomatic complexity for the anonymous class is added to the cyclomatic complexity of the containing method.

The Java archive (JAR) files resulting from the compilation of the source code were analysed with FindBugs. FindBugs employs a heuristic to determine the severity of the defects it finds and, in its default mode, issues *priority one* and *priority two* warnings, with priority one deemed the more serious. Counts of priority one and priority two warnings were recorded for each class and method. The default settings for FindBugs were used with the exception of a filter to exclude warnings of the use of unconventional capitalisation of the first letter in class, method and field names, which would overlap with the findings of my tool. I also filtered out the "Dead Local Store" warning, which may result from the actions of the Java compiler.

The identifier naming and metrics data collected for each Java class and method was stored in XML files and collated with the XML output of FindBugs and the readability metric tool, using a tool I developed. Data extracted from the source code was matched with classes recorded by FindBugs to ensure that only identifiers from classes compiled into the JAR files were analysed. The collated data for each method was then written to R (R Development Core Team, 2008) dataframes for statistical analysis.

2.4.2 Statistical Analysis

The non-parametric chi-square test and Fisher's Exact test (Crawley, 2005) are used to determine whether any association existed between the presence of FindBugs warnings and identifier flaws in classes and methods, and between flawed identifiers and readability, cyclomatic complexity and MI in methods.

In the case of classes, contingency tables were created for each type of identifier flaw and FindBugs warning by determining the number of classes containing both the warning and identifier flaw, the number of classes with the warning only, those with the flaw only, and those without both the warning and the flaw. Table 2.3 shows the contingency table for priority two warnings and Long Identifier Name flaws in Tomcat classes.

Table 2.3: Example Contingency Table

Tomcat		**FindBugs Priority Two Warnings**	
		classes with	**classes without**
Long Identifiers	**classes with**	122	127
	classes without	165	605

Before applying either statistical test a contingency table of expected values is created. The expected value for each cell is calculated by multiplying the row total by the column total and dividing by the total number of classes, e.g. for Table 2.3 the expected value for the top left cell is $(122 + 165) * (122 + 127)/(122 + 165 + 127 + 605) = 70.13$

The chi-square test, with Yates' continuity correction, was applied to each contingency table, with the null hypothesis that the observed frequencies of warnings and flaws were independent. The Yates' continuity correction was used as it gives more conservative p-values than the plain chi-square test. Where any of the expected contingencies were less than five, I applied Fisher's Exact Test with the null hypothesis that the odds ratio was equal to one. The odds ratio shows the directionality of any relationship, with a value of one showing no relationship. In the case of Cactus, for which FindBugs reported a total of 20 defects in 128 classes, Fisher's Exact Test was often the only analytical method that could be applied.

For each test where the p-value was less than 0.05, the contingency table was compared with the table of expected frequencies to establish the nature of the association. For Table 2.3

p is less than 0.05 and the observed frequency of 122 for the top left cell is greater than the expected value of 70.13. Thus, for Tomcat, there is a statistically significant association between priority two warnings and the presence of Long Identifier Name flaws in classes.

In addition to the chi-square tests when analysing methods, I applied another technique to analyse contingency tables that is used to evaluate diagnostic tests to determine whether the observed phenomena have a practical application. The same contingency tables used for the chi-square tests were analysed by treating FindBugs warnings, the maintainability index, cyclomatic complexity and readability as reference classifiers. For example, for the contingency table in Table 2.4 the occurrence of FindBugs priority two warnings in methods is treated as the reference classifier, and the performance of the Non-Dictionary Words flaw as a classifier is tested in comparison.

Table 2.4: Example Contingency Table

JFreeChart		**FindBugs Priority Two Warnings**	
		methods with	**methods without**
Non-Dictionary Words	**methods with**	103	2925
	methods without	37	5165

To evaluate the relative performance of the test classifier, two quantities are derived from the contingency table: the *sensitivity* and the *specificity*, which represent agreement between the two classifiers. The sensitivity is the proportion of the population classified as positive by the reference classifier that are classified positively by the classifier being tested. In the example in Table 2.4, the sensitivity is the proportion of methods for which FindBugs warnings are issued, that also contain identifiers with the Non-Dictionary Words flaw; i.e. $sensitivity = 103 \div (103 + 37) = 0.74$. The specificity is the proportion of population classified negatively by the reference classifier that are also classified negatively by the test classifier. In Table 2.4, the specificity is the proportion of the methods without FindBugs priority two warnings that have no identifiers with the Non-Dictionary Words flaw: $specificity = 5165 \div (2925 + 5165) = 0.64$. An advantage of this method is that sensitivity and specificity are independent of the rate of incidence, or prevalence, of the phenomenon being investigated.

The characteristics of a given test can be illustrated using receiver operating characteristic (ROC) curves, where the *sensitivity* of a test is plotted on the y-axis, against $1 - specificity$

(the false positive rate) on the x-axis. The area under the curve (AUC) (see Figure 2.1) indicates the efficacy of the test. A useless test, i.e. one that is equivalent to guessing, is indicated by a diagonal line drawn from the origin to the top-right corner, representing the equation $sensitivity = 1 - specificity$, which has an AUC of 0.5. For a test to be useful the points plotted should lie above and to the left of the diagonal line. ROC curves are, thus, a means of visualising the predictive power of the observed associations.

The example results in a point at $(0.36, 0.74)$, above and to the left of the diagonal, meaning that, in the case of JFreeChart, using the Non-Dictionary Word flaw as a binary classifier is a *better than chance* method of predicting the presence or absence of FindBugs priority two warnings. The predictive power of a result is related to its perpendicular distance from the diagonal line, and is equal to the area under a line drawn from the origin to the point representing the result and from the result to the point $(1, 1)$. In the example, the predictive power is 0.69, which means that the Non-Dictionary Word flaw has a 0.69 probability of indicating whether or not a method contains a FindBugs Priority two warning in JFreeChart.

The majority of methods in JFreeChart are correctly classified by the test classifier and are grouped in the top-left and bottom-right cells of Table 2.4. As will be seen in the next section, especially for Cactus, it is possible for the members of a population to be grouped in these cells, resulting in values of *sensitivity* and *specificity* that give a useful probability, without the distribution in the contingency table giving a statistically significant result for either the chi-square test or Fisher's exact test.

2.4.3 Threats to Validity

Construct Validity The definition of the Short Identifier Name guideline is much more restrictive than the Java naming conventions (Sun Microsystems, 1999; Gosling *et al.*, 2014; Vermeulen *et al.*, 2000) and common practice. Consequently the number of identifiers categorised as flawed may be inflated, and accordingly the observed associations may need to be treated with caution.

FindBugs, like other static analysis tools, reports false positives (Ayewah *et al.*, 2007). Without inspecting the source code for all warnings, the false positive rate cannot be established. Another concern is that FindBugs may have been applied to the subject code bases. In response to my enquiries, the developers of FreeMind, jEdit and JFreeChart have indicated

that they do not use FindBugs systematically, if at all.

External Validity The apparently project-specific influences on the relationships between flawed identifiers and FindBugs warnings in Table 2.6, suggest that, though general principles may be derived from my findings, caution is necessary when applying them to other projects. Some project-specific variation is apparent even in the more consistent findings shown in Tables 2.7 and 2.8, again suggesting that care may be required when applying these findings.

2.5 Results

This section reports the result of the class and method analyses. The identifier flaws Consecutive Underscores and Enumeration Identifier Declaration Order (Table 2.1) are excluded from the statistical analysis because the former was not found in any of the code bases, and the latter only very rarely.

2.5.1 Class Analysis

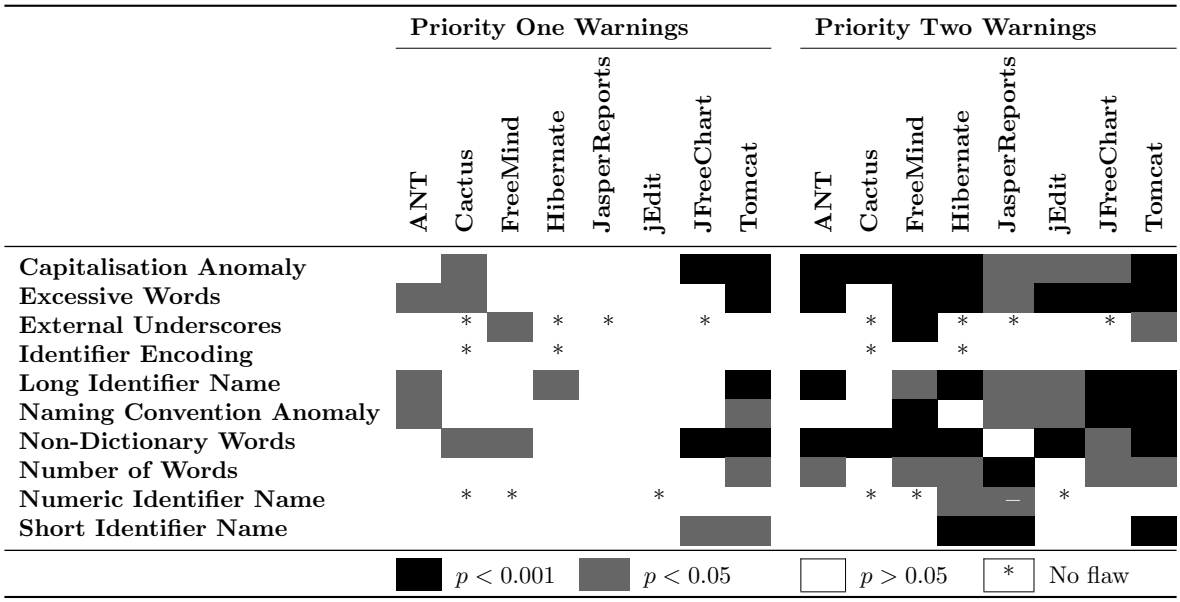
Table 2.5 shows the statistically significant associations between identifier flaws and FindBugs warnings in dark grey for $p < 0.05$ and black for $p < 0.001$, the absence of any significant association (i.e. $p > 0.05$) in light grey, and blank areas show the absence of the identifier flaw[1]. For each statistically significant relationship, the trend was checked and the observed frequency of the occurrence of identifier flaws and FindBugs Warnings together in classes was always greater than expected by chance, with one exception (marked with '–'), which I discuss later.

Table 2.5 shows that associations between priority one warnings and identifier flaws are less common than the more consistent associations for priority two warnings. The Capitalisation Anomaly, Dictionary Words, Excessive Words and Long Identifier Name flaws are each associated with priority two warnings in at least seven of the eight code bases.

Statistically significant relationships between identifier flaws and defects for Cactus in Table 2.5 are constrained to the Capitalisation Anomaly, Dictionary Words and Excessive Words flaws. A possible explanation is that the Cactus development team use CheckStyle (Burn, 2007) to ensure committed code conforms to the project's detailed coding conventions

[1]Full details of the chi-square values can be found at `http://www.facetus.org.uk/conferences/WCRE09/`

Table 2.5: Associations Between Naming Flaws and Priority One and Two Warnings at Class Level



(Apache Software Foundation, 2008). Their development methodology reduces the number of flawed identifiers; however, my tool analyses capitalisation more rigorously than CheckStyle, and performs checks for the length of identifiers and natural language content, which CheckStyle does not.

The only statistically significant negative association found (marked '–') is between Numeric Identifier Name flaws and priority two defects for JasperReports. Examination of the extracted identifiers found only the constants `ZERO` and `ONE` composed of numeric words alone, and that the former is used in 50 classes without FindBugs warnings.

2.5.2 Method Analysis

In Tables 2.6, 2.7 and 2.8 statistically significant associations between the flawed identifiers and each of the source code quality measures are represented in black where $p < 0.001$ and dark grey where $p < 0.05$. Where the trend of association was negative, i.e. the presence of the particular identifier flaw is associated with better quality source code, the cell is marked with a white dash. White cells represent the lack of a statistically significant association (i.e. $p > 0.05$), and asterisks indicate where the particular identifier flaw was not found. The digits contained in selected cells show the probability with which the identifier flaw, when applied as a binary classifier, correctly predicts the quality of methods. Only probabilities

of 0.55, marginally better than guessing, or greater, have been included in the tables. The probabilities not shown are largely close to 0.5, and only less than 0.5 for some of the negative associations.

Table 2.6: Associations Between Naming Flaws and Methods Containing Priority One and Two Warnings

	Priority One Warnings								**Priority Two Warnings**							
	ANT	**Cactus**	**FreeMind**	**Hibernate**	**JasperReports**	**jEdit**	**JFreeChart**	**Tomcat**	**ANT**	**Cactus**	**FreeMind**	**Hibernate**	**JasperReports**	**jEdit**	**JFreeChart**	**Tomcat**
Capitalisation Anomaly	.71			.63	.59			.56	.62		.62	–	–			.57
Excessive Words			.55						.55	.55		.58	–			
External Underscores		*		*	*		*			*		*	*		*	
Long Identifier					.59					.59		.57	–			
Naming Convention Anomaly																
Number of Words	.57	.61	.62	.62	.64	.56	.59	.55	.56		.59	–			.55	.55
Numeric Identifier	.55	*	*			*		*		*	*			*		*
Short Identifier Name	.59	.64	.63	.65	.66		.61	.59	.56	.58	.62	–			.56	.57
Type Encoding		*		*				*		*		*				*
Non-Dictionary Words	.72	.92	.71	.70	.66	.60	.81	.57	.60	.64	.62		–	.63	.69	.59
Extended 3	.71	.94	.66	.81				.55	.64	.66	.59			.63		.59
Extended 5	.76	.94	.66	.80		.57	.88	.56	.64	.65	.64	–		.63	.72	.59
Extended 10	.72	.92	.65	.75		.67	.87	.55	.63	.64	.64	–		.61	.72	.61
Less-readable	.82	.74	.72	–	.65		.72	.60	.67		.67	.67	–		.66	.68

$p < 0.001$ $p < 0.05$ $p >= 0.05$ * No flaw

Each table lists three further categories labelled 'Extended 3', 'Extended 5' and 'Extended 10'. The results for the three 'Extended' flaws should be compared with those for the Non-Dictionary Words flaw to determine the influence of project-specific words and abbreviations on the relationship between the linguistic content of identifiers and FindBugs warnings. The bottom line of Table 2.6 shows the relationships between methods classified as less-readable by the readability metric and FindBugs warnings. Where associations were found associations, the results largely confirm the connection between readability and FindBugs warnings found by Buse and Weimer (2008). Indeed, my results show the connection between readability and FindBugs warnings extends to projects such as ANT and FreeMind, which Buse and Weimer did not investigate. However, this work differs in the statistical methods used, the versions of projects investigated, and because it discriminates between priority one and two warnings, which Buse and Weimer did not.

Table 2.6 shows the associations between identifier flaws and FindBugs priority one and

priority two warnings in the methods of each project. The statistical associations are largely confined to particular identifier flaws indicating the general cross-project trends. However, there are also apparent project-specific relationships as illustrated by Cactus and jEdit for both priority one warnings, and Cactus, Hibernate and JasperReports for priority two warnings.

While Cactus and jEdit have just one statistically significant association with priority one warnings between them, useful predictive qualities were found in the relationships for some identifier flaws. The probabilities given in the left hand side of Table 2.6 emphasise the cross-project nature of the relationships between the Extended, Non-Dictionary Words, Number of Words and Short Identifier flaws. The relationships for the priority two warnings are less clear. There are hints of similar, general, cross-project relationships; however, the project-specific relationships are more apparent. Cactus, again, has few statistical associations, but some relationships have probabilities greater than 0.55. Hibernate and JasperReports both have negative statistical associations. Hibernate has a few relationships with probabilities greater than 0.55, whereas JasperReports has none.

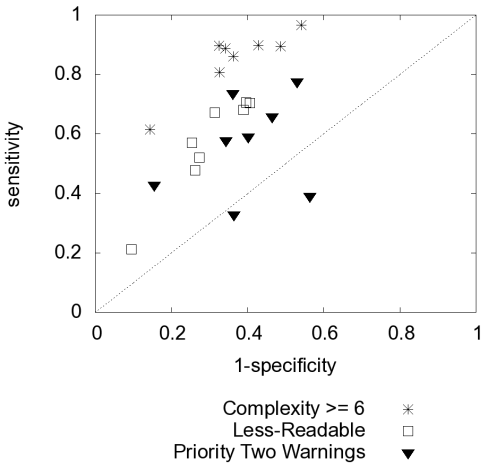


Figure 2.1: ROC Plot for the Non-Dictionary Words Flaw at the Method Level

The relationships for the Non-Dictionary Words flaw and priority two warnings are plotted in Figure 2.1. While six points are above the diagonal line and illustrate the utility of the Non-Dictionary Words flaw as a light-weight classifier, there are two points below the line. The point for Hibernate, where no statistically significant association was found, is closer to the line and the other is for JasperReports which has a negative association.

Tables 2.7 and 2.8 show much more consistent relationships for identifier flaws with complexity, maintainability and readability. There remain, however, hints of project-specific relationships, which are most apparent for Cactus. The predictive probability associated with each relationship illustrates the utility of the identifier flaws as light-weight classifiers for source code quality. The relationships between the Non-Dictionary Words flaw and complexity and readability are plotted in Figure 2.1.

Table 2.7: Associations Between Naming Flaws and Cyclomatic Complexity at the Method Level

	Cyclomatic Complexity >= 6								**Cyclomatic Complexity** >= 10							
	ANT	**Cactus**	**FreeMind**	**Hibernate**	**JasperReports**	**jEdit**	**JFreeChart**	**Tomcat**	**ANT**	**Cactus**	**FreeMind**	**Hibernate**	**JasperReports**	**jEdit**	**JFreeChart**	**Tomcat**
Capitalisation Anomaly	.68	.68	.65	.65	.65	.61	.68	.73	.67	.72	.63	.64	.66	.61	.73	.75
Excessive Words			.58	.62	.55		.58		.55	.55	.58	.65	.58		.60	
External Underscores		*		*	*		*			*		*	*		*	
Long Identifier		.56	.56	.64	.62		.57	.55		.56	.57	.68	.66		.58	.57
Naming Convention Anomaly														.55		
Number of Words	.56	.62	.57	.61	.65		.59	.60	.55	.61	.57	.60	.64		.58	.59
Numeric Identifier		*	*			*		*		*	*			*		*
Short Identifier Name	.58	.65	.58	.64	.64	.55	.61	.61	.63	.65	.57	.62	.62	.55	.60	.62
Type Encoding		*		*				*		*		*				*
Non-Dictionary Words	.70	.65	.68	.75	.69	.64	.78	.74	.67	.70	.67	.74	.70	.64	.78	.76
Extended 3	.69	.65	.63	.69	.65	.62	.69	.72	.69	.70	.61	.73	.68	.64	.75	.75
Extended 5	.71	.64	.65	.72	.69	.64	.80	.73	.70	.69	.65	.75	.73	.66	.82	.76
Extended 10	.71	.65	.66	.74	.70	.65	.80	.74	.70	.70	.66	.76	.74	.66	.81	.77

$p < 0.001$ $p < 0.05$ $p >= 0.05$ * No flaw

Since the investigations were completed, Posnett *et al.* (2011) have reviewed the work of Buse and Weimer (2008) and developed an improved readability metric, which relies on the Halstead Volume, LOC and software entropy. The model of readability developed by Posnett *et al.* is used as a benchmark by Börstler *et al.* (2015) to evaluate a readability metric created using an alternative approach derived from the Flesch Reading Ease Score (Flesch, 1948). Börstler *et al.* report similarity between their source code readability metric and that developed by Posnett *et al.*. Of interest is that Börstler *et al.* also report close correlation with the Maintainability Index (Welker *et al.*, 1997), which arises because of the reliance of both metrics on the Halstead Volume and LOC metrics. Table 2.8 may be viewed, with this relationship in mind, using the Maintainability Index as approximation of the more recent readability metrics.

Table 2.8: Associations Between Naming Flaws and Readability and the Maintainability Index at the Method Level

	Less-Readable								Less-Maintainable							
	ANT	**Cactus**	**FreeMind**	**Hibernate**	**JasperReports**	**jEdit**	**JFreeChart**	**Tomcat**	**ANT**	**Cactus**	**FreeMind**	**Hibernate**	**JasperReports**	**jEdit**	**JFreeChart**	**Tomcat**
Capitalisation Anomaly	.62	.55	.61	.60	.62	.62	.63	.66	.78	.78	.76	.67	.67	.64	.81	.77
Excessive Words			.59	.58	.61		.57		.59	.58	.67	.68	.62	.57	.63	.55
External Underscores		*		*	*		*			*		*	*	.57	*	
Long Identifier		.56	.58	.60	.58		.56	.56	.57	.68	.67	.73	.71	.57	.61	.58
Naming Convention Anomaly									.55			.57	.56	.55		
Number of Words			.56		.60			.55	.57	.61	.62	.62	.65	.56	.59	.60
Numeric Identifier		*	*			*		*		*	*			*		*
Short Identifier Name								.57	.59	.65	.62	.65	.66	.56	.61	.63
Type Encoding		*		*				*		*		*				*
Non-Dictionary Words	.65	.56	.61	.66	.65	.65	.62	.68	.76	.77	.79	.82	.72	.72	.80	.78
Extended 3	.62		.56	.58		.62	.60	.65	.81	.76	.69	.83	.72	.71	.84	.80
Extended 5	.64		.57	.60		.63	.63	.66	.82	.76	.75	.85	.78	.74	.85	.80
Extended 10	.65	.56	.58	.63		.65	.63	.68	.80	.77	.77	.85	.80	.74	.84	.80

$p < 0.001$ $p < 0.05$ $p >= 0.05$ * No flaw

2.6 Discussion

Table 2.5 shows that priority one warnings often occur independently of identifier flaws where a class is analysed. This suggests programmers are capable of making significant errors irrespective of the degree of adherence to naming conventions. However, that strong associations often exist at the class level between priority two warnings and particular identifier flaws indicates connections between the use of low quality identifiers and less serious FindBugs defects.

The statistically significant associations found for FindBugs priority one and two warnings and identifier name flaws at the method level (Table 2.6) contain common features. There appear to be general, cross-project associations for some identifier flaws, but the distribution of associations appears to be largely project specific. Cactus is the most extreme example with statistically significant associations found with the chi-square test and Fisher's exact test only between the extended dictionaries and priority two warnings. jEdit has only one statistically significant association with priority one warnings, but more with priority two. The negative associations in Table 2.6 (marked with white dashes) emphasise the project specific nature of some relationships. That the negative associations are positive for the more serious priority one warnings, suggests that the developers in both projects may face

more complex issues with identifiers than can be explained without further investigation.

Fewer relationships between identifier flaws and priority one warnings, and more general relationships with priority two warnings were found at the class level than at the method level. At the method level a proportion of FindBugs warnings, which apply only to classes, are eliminated from the investigation. The finer-grained analysis could be the sole explanatory factor for the difference between the two sets of results for FindBugs warnings. However, it is possible that FindBugs warnings applicable at the class level alone, may have been a source of noise.

The negative associations for the Excessive Words and Long Identifier flaws for methods in JasperReports (Table 2.6) may be connected through the widespread use of longer identifier names, with which the development team have become familiar. The negative association for the Non-Dictionary Word flaw is not found with the lower frequency extended dictionaries and becomes a positive association with the 'Extended 10' flaw, indicating the importance of widely used application-specific terms in JasperReports. The use of application-specific terms is consistent with the commercialised nature of JasperReports and the finding of Lawrie *et al.* (2007b) that domain-specific natural language and abbreviations are more common in identifiers found in commercial source code than in open source. It is notable that the association between identifiers constructed of non-dictionary words and priority two defects is statistically significant at the class level in all but one of the subject code bases (JasperReports, Table 2.5) because empirical studies by Lawrie *et al.* (2006) have shown that the use of dictionary words makes identifiers easier to read and understand.

The evaluation of the predictive quality of each relationship at the method level offers further insights. Some relationships, despite the statistical independence of the two classifiers, may be applied as heuristics. The Non-Dictionary Word flaw for Cactus, for example, could be applied as reasonably reliable classifier of source code for FindBugs priority one warnings, with a probability of >0.9. In general, the Non-Dictionary Words flaw is a fair to good classifier for FindBugs warnings; however, it is not perfect. The Number of Words and Short Identifiers flaws are much weaker classifiers, with probabilities largely between 0.55 and 0.60, but are still better than guessing.

Tables 2.7 and 2.8 show largely consistent associations between the presence of identifier flaws and lower quality source code. In both cases the Capitalisation Anomaly and Non-

Dictionary Words flaws provide the stronger classifiers. For complexity and maintainability the Excessive Words, Long Identifier Name, Number of Words, and Short Identifier Name flaws also perform better than chance. However, only the Capitalisation Anomaly and Non-Dictionary Words flaws have consistent relationships with readability.

Identifier length is the only characteristic of individual identifiers that is a component of the readability metric. However, Buse and Weimer (2008) found that identifier length was not a significant influence on the readability of source code. The findings, shown in the left hand side of Table 2.8, suggest the human subjects, against whose judgements of source code readability the metric was trained, were influenced by the conformance of identifier names to familiar typographical conventions, and the use of dictionary words and well-known abbreviations. Further, the findings suggest that longer identifiers do have a negative influence on readability, as evidenced by the statistical associations found for the Excessive Words and Long Identifier flaws in Table 2.8.

The ROC plots for the Non-Dictionary Words flaw in Figure 2.1 illustrate that the flaw may be applied to predict lower quality source code. Tables 2.7 and 2.8 record probabilities generally greater than 0.6 and sometimes as high as 0.8, showing that the Non-Dictionary Words flaw provides a usable, light-weight classifier for the complexity, maintainability and readability of source code. The probabilities for other identifier flaws given in Tables 2.7 and 2.8 show similar predictive values for identifying less-readable, less maintainable and more complex source code. However, the probabilities given in Table 2.6 show that identifier flaws can be a useful, but not always reliable, predictor of FindBugs warnings because of the variation in the strength of associations between projects. Section 2.5.1 reports that the Cactus project requires the use of static style checking before code is committed to version control, which influences identifier quality. Also, the commercialised nature of the Hibernate and JasperReports projects may influence the composition of their identifiers (Lawrie *et al.*, 2007b). Boogerd and Moonen (2008, 2009) attributed many of the differences in their studies to 'domain factors'. As I deliberately chose not to include projects from identical domains, the results cannot offer clear conclusions on this question.

The naming guidelines used in this investigation are simple assessments of name quality. For example, the Non Dictionary words flaw does not consider whether the words used are in a coherent order, something that is important to those reading source code, and for the

automated extraction of information from names. Inspection of the names extracted from source code showed where developers break the guidelines they can do so systematically, which suggests that some developers have a notion of identifier name quality aspects of which differ from that tested. Some field names, for example, have a single letter prefix '`f`' that indicates the identifier's role, something that Relf's guidelines and common naming conventions (Gosling *et al.*, 2014; Vermeulen *et al.*, 2000) do not consider. I also found names that are not the simple noun and verb phrases advocated by the naming conventions, for example `popupMenuWillBecomeInvisible` (JUnit). Liblit *et al.*'s identification of phrasal structure in identifier names (Liblit *et al.*, 2006) identify only the shorter, single phrase names that, largely, agree with the recommendations of naming conventions. While longer names with more complex structures appear to be uncommon, the developers have felt the need to use them. Høst and Østvold (2008, 2009) investigated Java method names in detail and found that while developers use the phrasal structures specified in naming conventions in method names, they also use more complex phrasal structures. Høst and Østvold's work, in combination with my observations of names during the investigation reported in this chapter lead me to form the hypothesis that *developers use content types, including natural language content, in Java class and reference identifier names in ways that are richer and more varied than those specified in naming conventions, and described in the academic literature.*

To investigate the hypothesis I seek to answer the principal research question:

> **"What types of content and phrasal structure do developers use in Java class and reference names?"**

During the investigation reported in this chapter two practical issues became apparent: the overhead of processing source code to extract names for analysis, and limitations to the name tokenisation technique used. Analysis was implemented as a batch process where names were extracted from the source code for a project with some metadata and then analysed. While the process was tolerable with eight projects of moderate size, it would not scale well to analysis of more and larger projects. The conservative tokeniser I had implemented was unable to split names without typographical word boundaries, such as `arraycopy` (JUnit), and names containing digits. To undertake more detailed analysis of names requires a more aggressive approach to tokenisation.

2.7 Summary

The literature establishes the importance of identifier naming to program comprehension (Rajlich and Wilde, 2002; Lawrie *et al.*, 2006). However, there have been few investigations of the relationship between identifier name quality and source code quality (Boogerd and Moonen, 2008). The contribution of this investigation is to provide a deeper understanding of this important but largely unexplored relationship.

In this chapter I investigated the relationship between a range of identifier naming flaws found in Java classes and methods in 8 open source projects, and explored the possible relationships with a range of source code quality measures to answer the research question, "What relationship exists between the occurrence of low quality identifier names and lower quality source code?" The initial investigation at class level showed a relationship between identifier naming flaws and FindBugs warnings, which motivated a finer-grained investigation at the method level. The method level investigation incorporated a wider variety of source code quality measures to gain a richer perspective and discriminate among potentially confounding factors. The quality of identifier names was evaluated using accepted naming conventions validated by empirical study (Relf, 2004), and the natural language content of identifiers, including Java- and project-specific terms. Source code quality was evaluated at the method level using four perspectives: the identification of potentially problematic code with FindBugs, the three-metric maintainability index, a human-trained readability metric, and cyclomatic complexity. The chi-square test and Fisher's exact test were used to test the independence of poor quality identifiers and more-complex, less-maintainable, and less-readable source code. I found, generally, that poor quality identifiers are associated with lower quality source code. To understand whether the observed associations might have a practical application, I applied a technique used to evaluate diagnostic tests. I found that some associations occurred with sufficient consistency that they could be applied in a practical setting to identify areas of source code as candidates for intelligent review. I also found that some relationships not found to be statistically significant with the chi-square test and Fisher's exact test were potentially useful classifiers.

During the investigation, I found names with content types and phrasal structures that are not specified in common naming conventions. Java method names have been extensively analysed, but class and reference names have not. I formulated a hypothesis and research

question which concerns the use of a richer variety of content types and phrasal structures in Java class and reference names than is suggested by naming conventions and the academic literature.

The following two chapters describe solutions to the limitations of the name extraction and tokenisation processes identified in the previous section. In Chapter 3 I describe the selection of a corpus of projects and the development of a system to extract and store identifier names to support my research, and Chapter 4 reports on the development of novel techniques to tokenise identifier names to make their content available for analysis. I return to the principal research question in Chapter 5 with an investigation of class names.

Chapter 3

Identifier Name Extraction and Storage

The method adopted for the studies reported in the preceding chapter processed and analysed the identifier names from each project as a batch. My intention in the research reported in the remainder of this dissertation is to study names: both those found within a single project and within a corpus of names of a given *species* (class, method, field, etc.) gathered from multiple projects. In this chapter I describe the design and implementation of a methodology to extract identifier names from a corpus of 60 FLOSS Java projects and store them so that they are available to analytical tools, with metadata, without the overhead of parsing large bodies of source code each time the analytical software is run.

This chapter outlines why existing source code metamodels do not provide a viable solution, before describing a new metamodel that records source code structure and exposes names for recovery. A database structure for recording the model is introduced and the tool used to mine source code to populate the database is explained. Identifier name tokenisation is an integral part of the process of creating the database and the development and evaluation of improved name tokenisation techniques is described in detail in the following chapter.

The resulting database of names, known as `INVocD` (*I*dentifier *N*ame *Voc*abulary *D*atabase), is the main data source for the research reported in Chapters 4, 5, 6 and 7 and is publicly available[1].

[1] `http://oro.open.ac.uk/36992/`

3.1 The Corpus

A corpus of 60 FLOSS Java projects was selected. The projects in the corpus include later versions of the 8 investigated in studies described in the previous chapter and a further fifty-two selected as examples of projects from a range of domains including graphics libraries, programming languages, project management applications, and servers, and where the output of compilation might be a GUI application, a command line application, or a library or framework intended to be part of another application. All 60 projects were developed in English. English identifier names were chosen because it is the most widely used natural language in software development. The full list of projects and their version is given in Appendix B. The projects were selected to avoid potential bias that might arise from analysing names in a single domain — either the application domain or the implementation domain. The projects also range in scale and origin from large projects such as the Java development kit (JDK) that are managed by large commercial organisations, through projects that are part of the Apache Software Foundation (ASF), including the Derby database and Maven, to smaller, independent and popular projects, such as the JUnit testing framework, and the Jin chess client. Together, the projects in the corpus consist of some 16.5 MLOC (million lines of code) of Java as measured by SLOCCount (Wheeler, 2008).

My research interest lies in analysing the names declared in the projects, and I need some means of storing and accessing the names in the corpus, and, ideally, that method should be efficient. The computational and temporal overhead of repeatedly parsing 60 projects to extract names is inefficient and the use of an intermediary form from which names and metadata can be easily extracted would better support the research. In the next section I consider some existing source code models to understand whether they can provide a practical solution to support my research.

3.2 Existing Metamodels of Source Code

Generic source code metamodels, e.g. the Dagstuhl middle model (DMM) (Lethbridge *et al.*, 2004) and FAMIX (Ducasse *et al.*, 2011), are AST based and record source code as a graph. Both models also include additional relationships between program entities that, for example, record method invocation and inheritance. Names are recorded in both models as attributes

of AST nodes. The names are not first class members of the metamodel and can be accessed only by traversing the AST nodes recorded in the model. Thus, for the researcher interested in names, DMM and FAMIX models require additional processing to extract names, and, accordingly, using such metamodels as a means of storing names would be less efficient than extracting names and metadata by parsing source code. A possible alternative is provided by srcML (Collard *et al.*, 2011). srcML is an XML representation of source code that includes AST information and may, like any XML document, be queried using XPath. Again, however, there is the overhead of extracting names and metadata from srcML which is similar to an AST. A further issue is that all three approaches are also intended to model individual source code projects, and thus storage of multiple projects would require additional work.

Having found that available source code metamodels do not provide an efficient solution to making the names from multiple projects available for analytical tools, I ask the following research question:

> **RQ 1 How can a model for source code be created where identifier names and named AST nodes are both first class citizens?**

3.3 Model Design and Implementation

There are two aspects of the problem that need to be addressed: the development of the model, and the implementation of a means of persistence. I first describe the model of source code that records identifier name declarations, associated metadata, the names and their tokens (I use the term *token* because not all components of names are words). I then outline a database schema to store the model so the model can be recovered, and that the database can be queried, for example, to provide all the local variable names in a project with their metadata, or all the names in the 60 projects that start with the word 'array'.

3.3.1 A Model for Source Code Vocabulary

To support my research, I developed a generic model of object-oriented source code and identifier name vocabulary, the Source Vocabulary Model (*SVM*). SVM provides sufficient structural metadata for source code entities to identify the origin of an identifier name, both in terms of the program entity names, and the program entity's location in the source code

file. The source code vocabulary, the identifier names and their tokens are preserved, and are cross referenced with every usage. Identifier names and their tokens are therefore first class citizens of the model and are made directly available to users of the model. Thus, the structural model of source code can be accessed through its vocabulary, an important feature for identifier name research. The structural metadata, for instance, supports the recovery of all class names from the model, and other metadata, such as access modifiers, supports the modelling of the scope of an identifier name.

The model of source code used in SVM (Figure 3.1) can be extracted from the AST and is predicated on the idea that a program consists of a set of program entities that may contain other program entities. In general, identifier names are declared as labels for program entities, and the majority of program entities are named. By recording program entities, their relationships and their associated identifier names, the model records the source code structure and location and relative relationships of the items containing vocabulary found in the program. Note that larger containers such as file, package/namespace and project are omitted from Figure 3.1 to simplify the diagram. These larger containers, and their relationships to program entities, are illustrated in the database model in Figure 3.2.

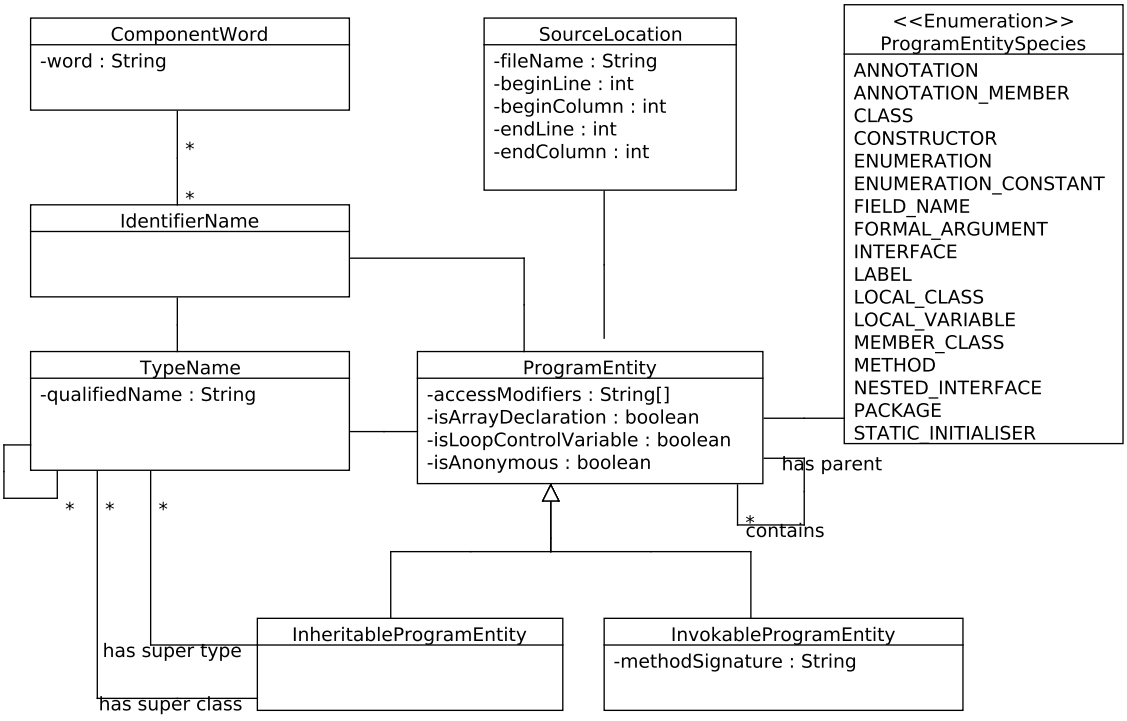


Figure 3.1: The SVM source code model

For a strongly typed programming language, such as Java, the core attributes of a program entity are its identifier name, type and species. Rather than specify a class for each species of program entity found in a programming language, an attribute is used to identify the species of program entity. In the specification of the SVM source code model in Figure 3.1 the species are Java specific and recorded in the enumeration `ProgramEntitySpecies`.

Most program entities also have one or more access modifiers that define their visibility to other parts of the program, and other properties. These generic characteristics are modelled as an attribute of the `ProgramEntity` class in Figure 3.1. The `ProgramEntity` class also references any comments associated with the program entity, information about the file containing the program entity and its location within the file, and a reference to the containing program entity.

In addition to these common characteristics, some program entities have specific characteristics. Classes and interfaces, for example, may be involved in inheritance, so, the super classes and super types, if any, need to be recorded. The requisite fields are provided by the `InheritableProgramEntity` subclass.

Many object-oriented programming languages allow constructors and methods to be overloaded by the declaration of multiple methods with the same identifier name, where each declaration has a parameter list composed of a unique sequence of types. Accordingly, constructors and methods require the addition of a signature listing the types of their formal arguments to identify them uniquely. Provision to distinguish between overloaded methods is made in the `InvokableProgramEntity` class, a subclass of `ProgramEntity`.

Type names are recorded using the `TypeName` class which contains fields to record the identifier name and the fully qualified name (*FQN*) (Gosling *et al.*, 2014) of the type. The FQN of a type is not always available in a given source code file, and may require additional information to resolve, such as the Java *classpath* used at the time of compilation. Accordingly, the FQN is a desirable attribute, but is not required. Parameterised type names, should there be any, are recorded through a reference to other `TypeName` objects. The distinction between parameterised types and type arguments in Java generics and C++ templates can be inferred from the context of the containing type name object. For example, in the class definition `ArrayList`⟨`E`⟩ in the `java.util` package `E` is a parameterised type, and in the variable declaration `ArrayList`⟨`String`⟩ `words`; `String` is a type argument;

I implemented SVM for the Java programming language in `JIM` (*J*ava *I*dentifier *M*iner), a tool that mines identifier names from Java source code and stores them, and the associated metadata, in an RDBMS. The remainder of this section describes the database schema and its implementation.

3.3.2 Database Schema

The database schema has two principal elements. The first is the `PROGRAM_ENTITIES` table that is used to store entities found on the AST. The second is the small group of tables that record the identifier name vocabulary. The two are linked, allowing the identifier name vocabulary used in program entities to be recovered, and the program entities in which particular words or groups of words are used to be identified, and their metadata recovered. (A complete list of the database tables and their columns is given in Appendix C.)

Rows in the `PROGRAM_ENTITIES` table represents each identifier name declaration in the AST and any unnamed nodes, such as initialiser blocks and anonymous classes, that might contain further identifier name declarations.

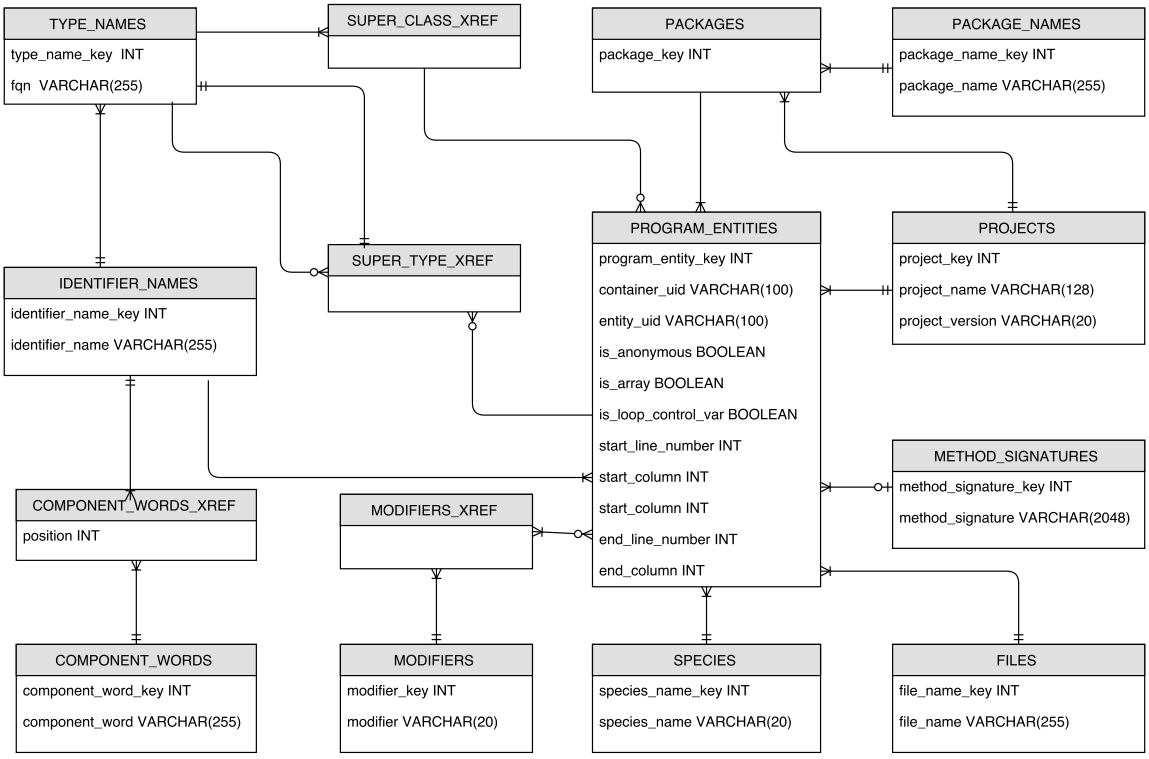


Figure 3.2: The INVocD database model

The PROGRAM_ENTITIES table contains the columns CONTAINER_UID and ENTITY_UID, which record the hierarchical relationships between program entities. The UIDs are generated during parsing using an SHA1 digest of a unique string for each file created by concatenating the project name and version, the package name and the file name. A serial number is then appended to the hash for each program entity encountered in the AST for that file. It is possible to use database keys instead of UIDs to record relationships, but that requires each container entity to be stored in the database before its children, which constrains the implementation of the system writing to the database.

For each constructor and method, a unique method signature is recorded in the METHOD_SIGNATURES table. The signature is a string composed of the type names of the formal arguments in declaration order. For example, the signature `(String;int)` is recorded for the `java.lang.String` method `indexOf(String str, int fromIndex)`. The signatures distinguish between multiple constructors, and methods of the same name, including overloaded methods.

The species of the program entity and Java *modifiers* (`private`, `static`, etc.) are recorded in separate tables. Species are stored in the SPECIES table and referenced from the PROGRAM_ENTITIES table. Modifiers are found in the MODIFIERS table and are cross-referenced through the MODIFIERS_XREF table, because there may be more than one modifier for each program entity. For example, fields that are class constants might be declared with the `public`, `final` and `static` modifiers.

The start and end line and column numbers of the program entity are recorded in the PROGRAM_ENTITIES table, and a foreign key references a unique file name recorded in the FILES table. File path information is not recorded, but may be derived, in part, from the package name (see Figure 3.3).

Packages and projects are also containers like program entities. However, they share little common data with the programming language constructs recorded as program entities, e.g. they do not have types or modifiers. Details of packages and projects are stored in separate tables. Projects are recorded as a name and version pair, which allows the database to store multiple versions of the same project. Packages are recorded as unique packages belonging to a specific project in the PACKAGES table, and unique package names are recorded in the PACKAGE_NAMES table. This approach saves space when storing multiple versions of projects,

```
select pn.package_name, f.file_name, pe.start_line_number
        from SVM.PROGRAM_ENTITIES pe
  join SVM.PACKAGES pkg
     on pkg.package_key=pe.package_key_fk
  join SVM.PACKAGE_NAMES pn
     on pn.package_name_key=pkg.package_name_key_fk
  join SVM.FILES f
     on f.file_name_key=pe.file_name_key_fk
  join SVM.IDENTIFIER_NAMES id
     on id.identifier_name_key=pe.identifier_name_key_fk
  join SVM.SPECIES sp
     on sp.species_name_key=pe.species_name_key_fk
  join SVM.PROJECTS pr
     on pr.project_key=pe.project_key_fk
  where pr.project_name='xom'
     and sp.species_name='method'
     and id.identifier_name='toString';
```

Figure 3.3: Example SQL query to identify the start locations of `toString`() method declarations in XOM

where the package names remain largely unchanged between versions.

Unique identifier names are stored in the `IDENTIFIER_NAMES` table, which is referenced by both program entities and type names. The component tokens of identifier names are recorded as unique tokens in the `COMPONENT_WORDS` table and linked to identifier names through a cross reference table (`COMPONENT_WORDS_XREFS`) that records the position of the token within the identifier name where the token is found (indexing starts at 1). For example, keys for the identifier name `ANEWARRAY` and the token `array` are stored in `COMPONENT_WORDS_XREFS` with the position 3. (An example SQL query using these tables is given in Figure 3.4.)

```
select identifier_name from SVM.IDENTIFIER_NAMES id
  join SVM.COMPONENT_WORDS_XREFS cwx
     on id.identifier_name_key=cwx.identifier_name_key_fk
  join SVM.COMPONENT_WORDS cw
     on cw.component_word_key=cwx.component_word_key_fk
  where cw.component_word='array' and cwx.position=1;
```

Figure 3.4: Example SQL query to recover unique identifier names beginning with the word 'array'

Type names are stored in the `TYPE_NAMES` table with a reference to the identifier name used to specify the type. The fully qualified name of the type may be recorded in the `TYPE_NAME`

column. The resolution of fully qualified type names is experimental and is discussed below. Generic types are not recorded in this database. Any type with a generic type argument is recorded as the underlying type and the type argument ignored, e.g. `ArrayList⟨String⟩` is recorded as `ArrayList`.

Type names involved in inheritance are cross referenced with program entities using two tables: `SUPER_CLASS_XREF`, which corresponds to class based inheritance (Java `extends` keyword), and `SUPER_TYPE_XREF` to type based inheritance (Java `implements` keyword, and `extends` when applied to interfaces).

Limitations

There are three limitations to the information stored in `INVocD`. The first is that the database schema was developed for the study of identifier names, not data flow or invocation, and hence records only identifier declarations, not usage. The second is the recording of qualified type names, which is experimental. The third is the processing of identifier names to extract vocabulary.

Typical Java builds are automated with tools like ANT and Maven that are configured with often complex build paths indicating where external Java classes and libraries are found, and external symbols may be resolved. My intention was to create a single pass tool that processed source code only. Hence, the resolution of external names is imprecise. External symbols in Java may be indicated explicitly in the source code in `import` statements, or implied in 'starred' imports where all the public entities of a package are made available to the Java compiler to resolve. Alternatively, an external symbol may be found in the local package, or the default `java.lang` package. The `TYPE_NAMES` table records a reference to the unique identifier name used to specify the type in the source code file. Where the fully qualified type name is given in an import statement, it is recorded, otherwise heuristics are used — such as knowledge of public names in the local package, and fully qualified types already seen that may match starred imports — to try to identify the fully qualified type name. Consequently, the fully qualified type name may not be present, and may not be reliable even if it is present.

The extraction of component words from identifier names is not trivial and the accuracy of the identifier name tokeniser is a limiting factor on the quality of the component word

vocabulary. INVocD was generated using my own tokeniser, `INTT`, which is described in the following chapter. `INTT` relies on dictionaries and can oversplit unrecognised abbreviations and words. Since `INVocD` includes the original identifier names, users can apply their own tokenisers.

Alternative Database Implementations

Consideration was given to using a NoSQL solution to provide the persistence layer, and particular attention was paid to the Neo4J graph database[2] given the graph structure of source code. Ultimately, an RDBMS was chosen in preference to a NoSQL solution because of familiarity with the technology and the wider availability of tools and libraries to access the databases. Neo4J has developed since the database was developed and now appears to offer as viable a solution as an RDBMS and may, in some circumstances, provide a more appropriate implementation of persistence.

3.3.3 Database Access

The database may be accessed using SQL queries either in a command line access tool, such as Derby's `ij`, or in a GUI tool. Both methods are useful for relatively simple queries. However, for larger analytical tasks, automation provides a much better solution. The tool used to compile the database, which is discussed in the following section, has a layered architecture the lowest layer of which handles database persistence. The persistence layer, known as `JIMdb`, was designed as a separate Java library so that it could be used by analytical tools to extract names and metadata from the database. The advantages of creating a separate database library are that analytical tools can be developed relatively quickly, and that the database implementation can also be changed.

3.4 Identifier Name Extraction

The tool used to create the database, `JIM`, consists of 3 main components — a coordinating layer, a parser layer and a persistence layer (`JIMdb`) — that in combination extract identifier names, component words and metadata concerning the location of the identifier name from

[2] `http://www.neo4j.org/`

source code and store it in an Apache Derby[3] database. Prior to storage identifier names are split into their component tokens using `INTT`, which is described in detail in the following chapter.

The coordination layer undertakes a recursive exploration of a directory tree supplied as a command line argument and parses every Java file found, with two exceptions: (1) files named `package-info.java` are ignored because they contain only package level documentation comments, not source code; and (2) files determined by heuristics to contain test code and generated code. However, `JIM` can be configured to parse generated and test code using command line arguments.

The parser layer consists of two parsers that reflect the introduction of new language constructs to the Java language in Java versions 1.5 and 1.7. The version of `JIM` used for much of this work contains a parser for Java version 1.5 and one for version 1.4 and earlier. Both parsers are based on the Sun Java 1.5 grammar supplied with the JavaCC parser generator[4]. Amongst the language features introduced in Java 1.5 was the new reserved word `enum` for enumerations. Consequently, source code that complies with older versions of Java and where `enum` is used as an identifier name will cause a Java 1.5 compliant parser to fail. Some modifications were made to the grammar to record identifier names in the resulting AST, as well as metadata such as access modifiers. For example, a production was introduced for identifier names that wrapped the identifier terminal making names easier to recover from the AST (See Figure 3.5). A parser for source files that conform to the Java 1.4 or earlier standard was created by removing keywords and productions related to features introduced in Java 1.5 from a second copy of the grammar. Substantive changes made to the Java programming language in Java 1.7 led to the the Java 1.5 parser being replaced with a parser generated from the Java 1.7 grammar published by the developers of ANTLR4[5]. Migration to ANTLR4 to parse Java 1.7 was necessary because no Java 1.7 grammar is available for JavaCC. (Further substantive changes to the Java language were introduced in Java 1.8. A Java 1.8 grammar is available for ANTLR4 and a Java 1.8 parser can be added to `JIM` to replace the Java 1.7 parser.)

Java initialiser blocks and anonymous classes are not named. Rather than store null

[3] `http://db.apache.org/derby/`
[4] `http://javacc.java.net/`
[5] Available from `https://github.com/antlr/grammars-v4`

```
void Identifier():
{ Token t; }
{
   t=<IDENTIFIER>
   { jjtThis.setText(t.image); }
}
```

Figure 3.5: JavaCC identifier terminal wrapped in a production

name in the database, I chose to record anonymous program entities with the identifier name '*#anonymous#*', which is not a legal Java identifier name. Similarly, program entities such as static initialisers that have no type are given the type name '*#no type#*'. This approach was initially adopted for debugging, but was found to be a useful convention that allows analytical tools that extract data from the database to identify unnamed and untyped program entities easily.

3.5 Summary

The SVM source code model devised in response to RQ 1, and its database implementation, offer advantages for identifier name research over existing source code metamodels. The chief advantage is that identifier names are first class citizens of the model and directly accessible from the database. Unlike the metamodels examined, the database model records multiple source code projects. By storing multiple projects in a single database, corpora of names extracted from more than one project may be studied, as can the names found in a single project. The database makes a practical contribution to the research reported in this dissertation because the computational and temporal overhead of repeatedly parsing multiple source code projects is removed. Consequently, the research effort can concentrate on the development of analytical tools. The creation of a database to provide a corpus for study using publicly available tools with documented methodologies, and where versions of the database are also publicly available, supports the replication of empirical studies.

Information extracted from SVM models recorded in the `INVocD` database is the data source for the study of Java class naming conventions in Chapter 5, and studies of Java reference names in Chapters 6 and 7. `JIM` has also been used to create databases of names used in an investigation of the relationship between identifier name vocabulary and the vocabulary used in change requests (Dilshener and Wermelinger, 2011).

The description of the design and creation of the `INVocD` database has not discussed identifier name tokenisation. The following chapter examines the problems encountered when splitting identifier names into their component tokens, and describes the solutions to those problems I developed and implemented in `INTT` to provide data for the analysis of class and reference names reported in remainder of the dissertation.

Chapter 4

Identifier Name Tokenisation

The previous chapter describes the design and creation of a system to extract and store identifier names, their tokens — i.e. the words, acronyms and abbreviations used to create the name — and metadata. The tokenisation of identifier names is undertaken by software engineering tools that process identifier names and is integral to the creation of the `INVocD` database. Identifier name tokenisation is, however, a non-trivial process. In this chapter I describe the practical problems encountered when trying to tokenise identifier names, including specific typographical features that have ambiguous tokenisations, and the absence of typographical features that mean, from a computational perspective, there is no obvious tokenisation. Specifically, I address the following two research questions:

> **RQ 2 How can more effective mechanisms to tokenise names with ambiguous or no word boundaries be developed?**; and
>
> **RQ 3 How can effective mechanisms to tokenise names containing digits be developed?**

I propose and implement solutions to each problem. The solutions are evaluated using sets of manually tokenised identifier names extracted from the corpus of 60 FLOSS Java projects described in the preceding chapter. The performance of the solutions is also evaluated against two existing methods.

The following section explains the difficulties encountered when tokenising identifier names, and some pitfalls potential solutions should avoid.

4.1 The Identifier Name Tokenisation Problem

Programming languages typically have typographical conventions that are employed by software developers to present source code so that it is readable. The typographical conventions typically include rules on bracket placement, indentation and identifier naming. In C, for example, function identifier names begin with a lower case character and consist of one or more words separated by the underscore character. In Java, method identifier names begin with a lower case character and consist of one or more words, and use internal capitalisation to mark the beginning of each word after the first. Visual Basic and C# method names follow a similar scheme to Java, but the first character is upper case. The typographical conventions are just that, conventions. For most programming languages the compiler pays no attention to typography, and so long as the identifier names used in source code are internally consistent there is no consequence for the compiled code. Consequently, the use of typographical conventions for identifier names is not compulsory, unless additional tools are used to police code at check-in time, and developers are free to create identifier names as they please.

A further issue, is that the typographical conventions were created with the intention of supporting source code readability. Conventions were not designed to support the extraction of information from identifier names, and do not support it as well as they might. So, while the typography of identifier names can help the extraction of information from identifier names, it is also an impediment, that may be misleading at times.

4.1.1 The Composition of Identifier Names

Programming languages and naming conventions constrain the content and form of identifier names. Programming languages impose hard constraints, most commonly that identifier names must consist of a single string[1], where the initial character is not a digit, and are composed of a restricted set of characters. For the majority of programming languages, the set of characters permitted in identifier names consists of upper and lower case letters, digits, and some additional characters used as separators. An additional hard constraint imposed by languages such as Perl and PHP is that identifier names begin with specific non-alphanumeric characters used as sigils — signs or symbols — to identify the type represented

[1]Smalltalk method names are a rare exception where the identifier name is separated to accommodate the arguments, e.g. `multiply: x by: y`

by the identifier. For example, in Perl '`$`' denotes a scalar and '`@`' a vector.

Naming conventions provide soft constraints in the form of rules on the parts of speech to be used in identifier names, how word boundaries should be constructed and often include the vague injunction that identifier names should be 'meaningful' (McConnell, 2004; Vermeulen *et al.*, 2000). Naming conventions typically advise developers to create identifier names with some typographical mechanism to identify boundaries between words. Java, for example, employs two conventions (Sun Microsystems, 1999): constants are composed of words and abbreviations in upper case characters and digits separated by underscores (e.g. `FOO_BAR`), and may be described by the regular expression $U[DU]^*(S[DU]^+)^*$, where D represents a digit, S a separator character and U an upper case letter; and all other identifier names rely on internal capitalisation to separate component words (e.g. `fooBar`).

4.1.2 Tokenising Identifier Names

Naming conventions, though applied widely, are soft constraints and, consequently, are not applied universally. Thus, tools that tokenise identifier names need to provide strategies for splitting both conventionally and unconventionally constructed identifier names. Identifier names contain features such as separator characters, changes in case, and digits that have an impact on tokenisation. I discuss each feature before looking at the difficulties encountered when attempting to tokenise identifier names without separator characters or changes in case to indicate word boundaries.

Separator Characters

Separator characters — for example, the hyphen in Lisp and the full-stop in R[2] — can be used to separate the component words in identifier names. Accordingly, the identification of conventional internal boundaries in identifier names is straightforward, and the vocabulary used by the creator of the identifier name can be recovered accurately.

Internal Capitalisation

Internal capitalisation, often referred to as *camel case*, is an alternative convention for marking word boundaries in identifier names. The start of the second and subsequent words in an iden-

[2] `http://www.r-project.org/`

tifier name are marked with an upper case letter as in the identifier name `StyledEditorKit` (found in the Java Library), where the boundary between the component words of an identifier name occurs at the transition between a lower case and an upper case letter, i.e. internally capitalised identifier names are of the form $U^?L^+(UL^+)^*$, where L represents a lower case letter, and the word boundary is characterised by the regular expression LU. The word boundary is easily detected and identifier names constructed using internal capitalisation are readily tokenised.

A second type of internal capitalisation boundary is found in practice. Some identifier names contain a sequence consisting of two or more upper case letters followed by at least one lower case letter, i.e. the sequence U^+UL^+. I refer to this type of boundary as the *UCLC boundary*, where UCLC is an abbreviation of upper case to lower case. Most commonly, identifier names with a UCLC boundary contain capitalised acronyms, for example the Java library class name `HTMLEditorKit`. In these cases the word boundary occurs after the penultimate upper case letter of the sequence. However, identifier names have also been found (Enslen *et al.*, 2009) with the same characteristic sequence where the word boundary is marked by the change of case from upper case to lower case, for example `PBinitialize` (Apache Derby). Thus, identification of the UCLC boundary alone is insufficient to support accurate tokenisation (Enslen *et al.*, 2009).

Some identifier names mix the internal capitalisation and separator character conventions, e.g. `ATTRIBUTE_fontSize` (JasperReports). Despite being unconventional, such identifier names pose no further problems for tokenisation than those already given.

Single Case

Some identifier names are composed exclusively of either upper case (U^+) or lower case characters (L^+), or are composed of a single upper case letter followed by lower case letters (UL^+). Such identifier names are often formed from a single word. However, some, such as `maxprefwidth` (Vuze) and `ALTORENDSTATE` (JDK), are composed of more than one word. Lacking word boundary markers, multi-word single case identifier names cannot be tokenised without the application of heuristics or the use of oracles. A variant of the single case pattern is also found within individual tokens in identifier names like `notAValueoutputstream` (Java library), where the developer has created a compound, or failed to mark word boundaries. Ac-

cordingly some tokens require inspection and, possibly, further tokenisation. When tokenising identifiers composed of a single case there are two dangers: *ambiguity* and *oversplitting*.

Ambiguity

Some single case identifier names have more than one possible tokenisation. For example, `ALTORENDSTATE` is, probably, intended to be interpreted as {`ALT`, `OR`, `END`, `STATE`}. However, it may also be tokenised as {`ALTO`, `RENDS`, `TATE`} by a greedy algorithm that recursively searches for the longest dictionary word match from the beginning of the string, leaving the proper noun 'Tate' as the remaining token. A function of tokenisation tools is therefore to disambiguate multiple tokenisations.

Oversplitting

The term *oversplitting* describes the excessive division of tokens by identifier name tokenisation software (Enslen *et al.*, 2009), e.g. tokenising the single case identifier name `outputfilename` as {`out`, `put`, `file`, `name`}. The consequence of this form of oversplitting is that search tools for concept location would not identify that 'output' was a component of `outputfilename` without additional effort to reconstruct words from tokens.

Oversplitting is also practised by developers in two forms: one conventional, the other unconventional. Oversplitting occurs in conventional practice in class identifier names that are part of an inheritance hierarchy. Class identifier names can be composed of part or all of the super class identifier name that may be consist of a number of tokens and an adjectival phrase indicating the specialisation. For example, the class identifier name `HTMLEditorKit` is composed of part of the type name of its super class `StyledEditorKit` and the adjectival acronym `HTML`, yet would be tokenised as {`HTML`, `Editor`, `Kit`}. In this case the compound of the super type is potentially lost, but can be recovered by program comprehension tools. Developers also oversplit components of identifier names unconventionally by inserting additional word boundaries, which increases the difficulty of recovering tokens that reflect the developer's intended meaning. Common instances include the oversplitting of tokens containing digits such as `Http_1_1`, the demarcation of some common prefixes as separate words as in `SubString`, and the division of some compounds such as *metadata* and *uppercase*. In each case, a recognisable semantic unit is subdivided into components and the composite meaning

is lost, and must be recovered by program comprehension tools (Ma *et al.*, 2008).

Digits

Digits occur in identifier names as part of an acronym or as discrete tokens. Where a digit or digits are embedded in the component word, as in the abbreviation `J2SE`, then the boundaries between tokens are defined by the internal capitalisation boundaries between the acronym and its neighbours. Abbreviations that have a bounding digit, e.g. `POP3` and `3D`, cannot be separated from other tokens where boundaries are defined by case transitions between alphabetical characters. Even if developers rigorously adopted the convention of only capitalising the initial character of acronyms advocated by Vermeulen *et al.* (2000), that would only help detect the boundary following a trailing digit (e.g. `Pop3Server`), it would not allow the assumption that a leading digit formed a boundary — that is it could not be assumed that UL^+DUL^+ may be tokenised as UL^+ and DUL^+. In other words, because digits do not appear in consistent positions in acronyms, there is no simple rule that can be applied to tokenise identifier names containing acronyms that include digits. Similar complications arise where digits form a discrete component of identifier names, including the use of digits as suffixes (e.g. `index3`) and as homophone substitutions for prepositions (e.g. `html2xml`).

The following section reviews the literature on identifier name tokenisation and the approaches adopted by other researchers to solving the problems outlined above.

4.2 Related Work

Although the tokenisation of identifier names is a relatively common activity undertaken by software engineering researchers (Abebe and Tonella, 2010; Antoniol *et al.*, 2002, 2007; Caprile and Tonella, 1999; Høst and Østvold, 2008; Kuhn *et al.*, 2007; Ma *et al.*, 2008; Marcus *et al.*, 2005; Nonnen *et al.*, 2011; Nonnen and Imhoff, 2012; Singer and Kirkham, 2008), few researchers evaluate and report their methodologies.

Feild *et al.* (2006) conducted an investigation of identifier name splitting. They used a conservative tokeniser — one that tokenises on separator characters and the lower to upper case transition — to tokenise names and focus their experimental effort on splitting single case identifier names, which they describe as *hard words*, into component, or *soft*, words. For example, the identifier name `hashtable` is constructed from the two soft words `hash` and

`table`.

Feild *et al.* compared three approaches to tokenising single case identifier names: a random algorithm, a greedy algorithm and a neural network. The greedy algorithm applied a recursive algorithm to match substrings of identifier names to words found in the ispell[3] dictionaries to identify potential soft words. For hard words that are not composed of a single soft word, the algorithm starts at the beginning and end of the string looking for the longest known word and repeats the process recursively for the remainder of the string. For example `outputfilename` is tokenised as {`output`, `filename`} from the beginning of the string and as {`outputfile`, `name`} from the end of the string on the first pass. The process is then repeated and the forward and backward components of the algorithm produce the same list of soft words, and thus the single tokenisation {`output`, `file`, `name`}. Where lists of soft words are different, the list containing the higher proportion of known soft words is selected.

Of the three approaches tried by Feild *et al.*, the neural network was the most accurate, but only under particular conditions, for example when the training set of tokenisations was created by an individual. However, its performance varied considerably (between 71% and 95% accuracy). The greedy algorithm performed most consistently in Feild *et al.*'s experiments, tokenising identifier names with an accuracy of 75–81%. The greedy algorithm, however, was prone to oversplitting names.

In a related study Lawrie *et al.* (2007b) turned to expanding abbreviations to support identifier name tokenisation, and posed the question: how should an ambiguous identifier name such as `thenewestone` be divided into component soft words? Depending on the algorithm used there are a number of plausible tokenisations and no obvious way of selecting the correct one, e.g. {`the`, `newest`, `one`}, {`then`, `ewe`, `stone`}, and {`then`, `ewes`, `tone`}. Lawrie *et al.* suggested that the solution lies in a heuristic that relies on the likelihood of the soft words being found in the vocabulary used in the program's identifier names.

Enslen *et al.* expanded on these ideas in a tool named *Samurai* (Enslen *et al.*, 2009). Samurai applies a four step algorithm to the tokenisation of identifier names.

1. Identifier names are first tokenised using boundaries marked by separator characters or the transitions between letters and digits.

2. The tokens from step 1 are investigated for the presence of changes from lower case to

[3] `http://www.gnu.org/software/ispell/ispell.html`

upper case (the primary internal capitalisation boundary) and split on those boundaries.

3. Tokens found to contain boundaries of the form U^+UL^+ – as in `HTMLEditor` – are investigated using an oracle to determine whether the token can be divided using the secondary internal capitalisation boundary, or whether the alternating case boundary offers a better tokenisation.

4. Each token is investigated using a recursive algorithm with the support of an oracle to determine whether it can be divided further.

The oracle used in steps 3 and 4 was constructed by recording the frequency of tokens resulting from naive tokenisation based on steps 1 and 2 found in identifier names extracted from 9,000 SourceForge projects. The oracle returns a score for a token based on its global frequency among all the code analysed and its frequency in the program being analysed. The algorithms in steps 3 and 4 are conservative. In step 3 the algorithm is biased to split on the internal capitalisation boundary, and will only split on the alternate case boundary where there is overwhelming evidence that the tokenisation is more frequent. The recursive algorithm applied in step 4 will only divide a single case string where there is strong evidence to do so, and also relies on lists of prefixes and suffixes to prevent oversplitting. For example, the token `listen` could be tokenised as {`list`, `en`} for projects where 'list' occurs as a token with much greater frequency than 'listen'. Samurai avoids such oversplitting by ignoring possible tokenisations where one of the candidate tokens, such as 'en', is found in the lists of prefixes and suffixes.

Enslen *et al.* also reproduced the 'greedy algorithm' reported by Feild *et al.* and compared the relative accuracies of the two techniques. The experiment used a 'gold' set of 8,000 identifier names that had been tokenised by hand. The Samurai algorithm performed better than their implementation of the greedy algorithm, with an accuracy of 97%. The Samurai algorithm has some limitations which I discuss in the next section.

The TIDIER algorithm (Madani *et al.*, 2010; Guerrouj *et al.*, 2012a) is derived from speech recognition techniques. Rather than relying on conventional internal capitalisation boundaries the approach tries to match substrings of an identifier name with entries in an oracle, both as a straightforward match and through a process of abbreviation expansion analogous to that used by a spell-checking program. Thus `idxcnt` would be tokenised as {`index`,

`count`}. Furthermore, because the algorithm ignores internal capitalisation it can consistently tokenise component words such as `MetaData` and `metadata`. Madani *et al.* achieved accuracy rates of between 93% and 96% in their evaluations, which was better than naive camel case splitting in both projects investigated. Geurrouj *et al.* found TIDIER to improve significantly on Samurai. The advantage appears to derive chiefly from abbreviation expansion, which Samurai does not do. The disadvantage of TIDIER is that it is processor intensive and does not provide a practical solution for tokenising names in desktop applications because its runtime for processing a single name is measured in minutes. Guerrouj *et al.* (2012b) also developed another approach, named TRIS, that uses domain vocabulary to support splitting names and expanding any abbreviations found. Their evaluation of TRIS shows that it is an improvement on TIDIER.

GenTest (Lawrie *et al.*, 2010) uses expansion of abbreviations and acronyms to help identify potential tokenisations which are evaluated using a large, proprietary, n-gram database. Normalize (Lawrie and Binkley, 2011) builds on GenTest by adding contextual vocabulary to identify the most likely of the proposed expansions, which identifies the more likely name tokenisation.

Identifier name tokenisation has been quite a competitive research area with many claims made concerning accuracy and the benefits of the various approaches and implementations. Hill *et al.* (2013) undertook an empirical study comparing the tokenisers described above, and `INTT` the tokeniser that I developed and describe below. A drawback of the individual studies described above is that each research group uses its own technique to evaluate the accuracy of their tokenisation method, which makes comparison of the methods difficult. Hill *et al.* created a reference set identifier name tokenisations (Binkley *et al.*, 2013) which they used to evaluate the performance of GenTest, `INTT`, Samurai and TIDIER. GenTest, `INTT` and Samurai performed best.

LINSEN (Corazza *et al.*, 2012) similarly seeks to identify potential tokenisations through the probability of likely expansions of abbreviations. Where LINSEN differs is that contextual information is gathered from comments in the file where the name is found, comments in the project, a technical dictionary and an English dictionary. Each dictionary is given respectively decreasing weight in terms of context. A graph of possible expansions and tokenisations is created, and the most likely chosen. LINSEN was evaluated and compared with GenTest,

Normalize and TIDIER, and was found to produce consistently more accurate tokenisations.

4.3 An Improved Approach

Of the solutions that existed when this research was undertaken — Samurai and TIDIER — each approach had been found to tokenise 96-97% of identifier names accurately. However, there are limitations to each solution and issues with their implementation that make their application in practical tools difficult. Of the approaches discussed, only Enslen *et al.* attempt to process identifier names containing digits. However, digits are isolated as separate tokens at an early stage of the Samurai algorithm so that meaningful acronyms such as `3d` and `http11` are tokenised as {`3`, `d`} and {`http`, `11`} respectively. While the latter case is a reasonable alternative to grouping the acronym and digits, splitting the former renders the acronym meaningless. Samurai is also hampered by the amount of data collection required to create its supporting oracle. TIDIER is processor intensive, and an impractical solution for desktop applications. A further problem was that neither solution was publicly available, so could not be used to support my research.

I implemented `INTT` to address the limitations in existing tools. In particular, I have tried to ensure that the solution is relatively easy to implement and deploy, and is able to tokenise all types of identifier name. `INTT` applies naive tokenisation to identifier names that contain conventional separator character and internal capitalisation word boundaries. Tokens containing the UCLC boundary or digits are processed using heuristics to determine a likely tokenisation, and identifier names composed of letters of a single case are tokenised using an evolution of the greedy algorithm, which is described in detail in Section 4.3.4.

The core tokenisation functionality of `INTT` is implemented in a JAR file so that it can be readily incorporated into other tools. The simple API allows the caller to invoke the tokeniser on a single string, and returns the tokens. Invoking applications can therefore range in sophistication from basic command line utilities that process individual identifier names to parser based tools that process source code. To support programming language independence the set of separator characters can be configured using the API, but the caller is responsible for removing any sigils from the identifier name. However, `INTT` has only been tested on identifier names extracted from Java source code.

In summary, my tokenisation algorithm consists of the following steps, which are discussed

in detail below:

1. Identifier names are tokenised using separator characters and the internal capitalisation boundaries.

2. Any token containing the UCLC boundary is tokenised with the support of an oracle.

3. Any identifier names with tokens containing digits are reviewed and tokenised using an oracle and a set of heuristics.

4. Any identifier name composed of a single token is investigated to determine whether it is a recognised word or a neologism constructed from the simple addition of known prefixes and suffixes to a recognised word.

5. Any remaining single token identifier names are tokenised by recursive algorithms. Candidate tokenisations are investigated to reduce oversplitting, before being scored with weight being given to tokens found in the project-specific vocabulary.

Steps 2, 4 and 5 form the answer to RQ 2, and step 3 provides the answer to RQ 3.

4.3.1 Oracles

To support the tokenisation of identifier names containing the UCLC boundary, digits and single case identifier names, I constructed three oracles: a list of dictionary words, a list of abbreviations and acronyms, and a list of acronyms containing digits. The list of dictionary words consists of some 117,000 words, including inflections and American and Canadian English spelling variations, from the SCOWL package word lists up to size 70, the largest lists consisting of words commonly found in published dictionaries (Atkinson, 2004). I added a further 120 common computing and Java terms, e.g. 'arity', 'hostname', 'symlink', and 'throwable'. The analysis reported in Chapter 2 included analysis of which identifier names did not correspond to dictionary words and found that several known computing terms were not found in commonly available dictionaries. The list of computing terms was hence constructed iteratively over the analysed projects, using the criterion that any word added should be a known, non-trivial computing term. Each oracle is implemented using a Java `HashSet` so that lookups are performed in constant time.

The use of dictionaries imposes a limitation on the accuracy of the resulting tokenisation because a natural language dictionary cannot be complete. I addressed this limitation by adopting a method to incorporate the lexicon of the program being processed in an additional oracle, which takes a step towards resolving the issue highlighted in Lawrie *et al.*'s question of how to resolve ambiguous tokenisations for identifier names such as `thenewestone` (Lawrie *et al.*, 2007b). Tokens resulting from the tokenisation of conventionally constructed identifier names are recorded in a temporary oracle to provide a local — i.e. domain- or project-specific — vocabulary that is employed to support the tokenisation of single case identifier names. For example, tokens extracted from identifier names such as `pageIdx` and `lineCnt` can be used to support the tokenisation of an identifier name like `idxcnt` as {`idx`, `cnt`}.

`INTT` is also able to incorporate alternative lists of dictionary words in its oracle, and is, thus, potentially language independent. `INTT` relies on Java's string and character representations, which default to the UTF-16 unicode character encoding standard. So, `INTT` is able to support dictionaries, and thus tokenise identifier names created using natural languages where all the characters, including accented characters, can be represented using UTF-16 (subject to the constraints on identifier name character sets imposed by the programming language). However, as `INTT` was designed with the English language and English morphology in mind, adaptation to other languages may not be straightforward.

4.3.2 Tokenising Conventionally Constructed Identifier Names (RQ 2)

The first stage of `INTT` tokenises identifier names using boundaries marked by separator characters and on the conventional lower case to upper case internal capitalisation boundaries. Where the UCLC boundary is identified, `INTT` investigates the two possible tokenistations: the conventional internal capitalisation where the boundary lies between the final two letters of the upper case sequence, e.g. as found in `HTMLEditorKit`; and the boundary following the sequence of upper case letters, as in `PBinitialize`. The preferred tokenisation is that containing more words found in the oracle. Where this is not a discriminant, tokenisation at the internal capitalisation boundary is preferred. Thus, part of the answer to RQ 2 is: where an ambiguous boundary is identified candidate tokenisations are created and evaluated.

Following the initial tokenisation process, identifier names are screened to identify those that require more detailed processing. Identifier names found to contain one or more tokens

with digits are tokenised using heuristics and an oracle. Identifier names composed of letters of a single case are tokenised, if necessary, using a variant of the greedy algorithm (Lawrie *et al.*, 2007b). These processes are described in detail below.

4.3.3 Tokenising Identifier Names Containing Digits (RQ 3)

Section 4.1 outlines the issues concerning the tokenisation of identifier names containing digits. I identified three uses of digits in identifier names: in acronyms (e.g. `getX500Principal` (JDK)), as suffixes (e.g. `typeList2` (JDK, Java libraries and Xerces)) and as homophone substitutes for prepositions (e.g. `ascii2binary` (JDK and Java libraries)). In the latter two cases the digit, or group of digits, forms a discrete token of the identifier, and if identified correctly the identifier name may be tokenised with relative ease. Acronyms containing digits are more problematic. I identified two basic forms of acronym: those with an embedded digit, e.g. `J2SE`, and those with one or more bounding digits, e.g. `3D`, `POP3` and `2of7`.

Acronyms with embedded digits are bounded by letters and can be tokenised correctly by relying on internal capitalisation boundaries alone. For example, the method identifier name `createJ2SEPlatform` (Netbeans) can be tokenised as as {`create`, `J2SE`, `Platform`} without any need to investigate the digit. Acronyms with leading or trailing digits cannot easily be tokenised, and neither can those bound by digits. I made a special case of acronyms with two bounding digits. While they could be tokenised on the assumption that the digits were discrete tokens, I decided that the very few instances of acronyms with two bounding digits found in the subject source code were better seen as discrete tokens for the purposes of program comprehension. Indeed all the instances found were noun phrases describing mappings, `1to1`, or bar code encoding schemes `2of7`.

With the exception of the embedded digit form of acronym there is no general rule by which to tokenise identifier names containing digits. Accordingly I created an oracle from a list of common acronyms containing digits and developed a set of heuristics to support the tokenisation of identifier names containing digits.

Identifier names are first tokenised using separator characters and the rules for internal capitalisation. Where a token is found to contain one or more digits it is investigated to determine whether it contains an acronym found in the oracle. Where the acronym is recognised the identifier name is tokenised so that the acronym is a token. For example, `Pop3StoreGBean`

can be tokenised using internal capitalisation as {`Pop3Store`, `G`, `Bean`}. The tokens are then investigated for known digit-containing acronyms and tokenised on the assumption that `Pop3` is a token, resulting in the tokenisation of the name as {`Pop3`, `Store`, `G`, `Bean`}.

Where known acronyms are not found, the digit-containing token is split to isolate the digit, or digits, and an attempt made to determine whether the digit is a suffix of the left hand textual fragment, a prefix of the right hand one, or a discrete token. I employ the following heuristics:

1. If the identifier name consists of a single token with a trailing digit, then the digit is a discrete token, e.g. `radius2` (Netbeans) is tokenised as {`radius`, `2`}.

2. If both the left and right hand tokens are both words or known acronyms the digit is assumed to be a suffix of the left hand token, e.g. `eclipse21Profile` (Eclipse) is tokenised as {`eclipse21`, `Profile`}.

3. If both the left and right hand tokens are unrecognised the digit is assumed to be a suffix of the left hand token, e.g. `c2tnb431r1` (Geronimo and JDK) is tokenised as {`c2`, `tnb431`, `r1`}.

4. If the left hand token is a known word and the right hand token is unrecognised, then the digit is assumed to be a prefix of the right hand token, e.g. `is9x` (Geronimo) is tokenised as {`is`, `9x`}.

5. If the digit is either a 2 or 4 and the left and right hand fragments are known words, the digit is assumed to be a homophone substitution for a preposition, and thus a discrete token, e.g. `ascii2binary` is tokenised as {`ascii`, `2`, `binary`}. It is trivial for the application that invokes the tokenisation tool to expand the digit into 'to' or 'for', if deemed relevant for the application.

4.3.4 Tokenising Single Case Identifier Names (RQ 2)

Tokenisation of single case names is related to the problem posed by the UCLC boundary which provides an implicit boundary that may be identified through comparison of two candidate tokenisations. With single case names there is no boundary implied by typography, instead possible and potential boundaries must be identified. To tokenise single case identifier

names I adapted the greedy algorithm developed by Feild *et al.* (2006). I identified two areas of the greedy algorithm that required improvement. Firstly, because the algorithm is greedy, it may fail to identify more accurate tokenisations in particular circumstances. For example, the algorithm finds the longest known word from beginning and end of the string, so Feild *et al.*'s example of `thenewestone` would be tokenised as {`then`, `ewes`, `tone`} by the forward pass, and as {`then`, `ewe`, `stone`} by the backward pass. Secondly, the algorithm assumes that the string to be processed begins or ends with a recognised soft word and therefore cannot locate soft words in a string that both begins and ends with unrecognised words or tokens.

My adaptation of the greedy algorithm is implemented in two forms: greedy and greedier. The names refer to the computational resources that may be consumed rather than any characteristic of the approach. The greedy algorithm assumes that the string being investigated either begins or ends with a known soft word and the greedier algorithm is only invoked when the greedy algorithm cannot tokenise the string.

Prior to the application of the greedy algorithm, strings are screened to ensure that they are not recognised words or simple neologisms. The check for simple neologisms uses lists of prefixes and suffixes[4] to check that strings are not composed of a combination of, for example, a known prefix followed by a known word. This allows identifier names consisting of Java neologisms such as `throwable` (Java Libraries, JDK and many more) to be recognised as words, despite them not being recorded in the dictionary used. The greedy algorithm iterates over the characters of the identifier name string forwards (see Algorithm 1) and backwards. On each iteration, the substring from the end of the string to the current character is tested using the dictionary words and acronyms oracles to establish whether the substring is a known word or acronym. When a match is found the soft word is stored in a list of candidates and the search invoked recursively on the remainder of the string. Where no word can be identified the remainder of the string is added to the list of candidates.

When the greedy algorithm is unable to tokenise the string, the greedier algorithm is invoked. The greedier algorithm attempts to tokenise a string by creating a prefix of increasing length from the initial characters and invokes the greedy algorithm on the remainder of the string to identify known words (see Algorithm 2). For example, for the string `cdoutputef`, `c` is

[4]The lists of prefixes and suffixes are based on those used in Samurai which are available from `https://hiper.cis.udel.edu/Samurai/`

Algorithm 1 `INTT` greedy algorithm: forward tokenisation pass

```
1: procedure greedyTokeniseForwards(s)
2:     candidates                                              ▷ a list of lists
3:     for i ← 0, length(s) do
4:         if s[0, i] is found in dictionary then
5:             rightCandidates ← greedyTokeniseForwards(s[i + 1, length(s)])
6:             for all lists of tokens in rightCandidates do
7:                 add s[0, i] to beginning of list
8:                 add list to candidates
9:             end for
10:        end if
11:    end for
12:    if candidates is empty then
13:        create new list with s as member
14:        add list to candidates
15:    end if
16:    return candidates
17: end procedure
```

added to a list of candidates and the greedy algorithm invoked on `doutputef`, then the prefix `cd` is tried and the greedy algorithm invoked on `outputef` resulting in the tokenisation {`cd`, `output`, `ef`}. This process is repeated, processing the string both forwards and backwards until the prefix and suffix are one character less than half the length of the string being tokenised, which allows the forward and backward passes to find small words sandwiched between long prefixes and suffixes, while avoiding redundant processing. For example in the string `yyytozzz` both the forwards and backwards passes will recognise `to`, and in the string `yyyytozz` the backwards pass will recognise `to`.

Algorithm 2 `INTT` greedier algorithm: backwards tokenisation pass

```
1: procedure greedierTokeniseBackwards(s)
2:     candidates                                              ▷ a list of lists
3:     for i ← length(s), length(s)/2 do
4:         leftCandidates ← greedyTokeniseBackwards(s[0, i − 1])
5:         for all lists of tokens in leftCandidates do
6:             add s[i, length(s)] to beginning of list
7:             add list to candidates
8:         end for
9:     end for
10:    return candidates
11: end procedure
```

Each list of candidate component words is scored according to the percentage of the

component words found in the dictionaries of words and abbreviations, and the program vocabulary — i.e. component words found in identifier names in the program that were split using conventional internal capitalisation boundaries and separator characters. The percentage of known words is recorded as an integer and a weight of one added for each word found in the program vocabulary. For example, suppose splitting `thenewestone` resulted in two candidate sets {`the`, `newest`, `one`} and {`then`, `ewe`, `stone`}. All the words in both sets are found in the dictionaries used and thus each set of candidates score 100. However, the words `newest` and `one` are found in the list of identifier names used in the program, so two is added to the score of the first set, and that is selected as the preferred tokenisation.

The algorithm, because of its intensive search for candidate component words, is prone to evaluating an oversplit tokenisation as a better option than a more plausible tokenisation. To reduce oversplitting, each candidate tokenisation is examined prior to scoring to determine whether adjacent soft words can be concatenated to form dictionary words. Where this is the case the oversplit set of tokens is replaced by the concatenated version. For example `outputfile` would be tokenised as {`output`, `file`} and {`out`, `put`, `file`}. Following the check for oversplitting, the first two tokens of the latter tokenisation would be concatenated making the two tokenisations identical, allowing one to be discarded.

A key advantage offered by the greedy and greedier algorithms are that a single case identifier name can be tokenised without the requirement that it begins or ends with a known word. For example, Feild *et al.*'s greedy algorithm cannot tokenise identifier names like `lboundsb` unless 'b' or 'l' are separate entries in the oracle. Samurai can only tokenise `lboundsb` if 'l' or 'lbounds' are found as separate tokens in the oracle. My algorithm can tokenise `lboundsb` using a dictionary where 'bounds' is an entry.

Part of the answer to RQ 2, given above, is that: "where an ambiguous boundary is identified candidate tokenisations are created and evaluated." Where single case tokenisations differ is that there is no readily identifiable ambiguous boundary. Two approaches are then used to search for possible token boundaries to complete the answer to RQ 2. First, the beginning and end of the string are treated as boundaries and the string searched to identify known words and potential boundaries, and create candidate tokens. Second, if a plausible tokenisation is not identified, the process is repeated by moving the point from which a search begins one character at a time from the beginning and end of the string. In both cases

candidate tokenisations are created and evaluated.

In the following section I evaluate the accuracy of my identifier name tokenisation algorithm and compare its performance with Samurai and Feild *et al.*'s greedy algorithm.

4.4 Experiments and Results

I adopted a similar approach to that used by Feild *et al.* (2006) and Enslen *et al.* (2009) to evaluate my approach and compare its performance with existing tools. However, instead of using a single test set of identifier names, I created seven test sets consisting of 4,000 identifier names each, extracted at random from the `INVocD` database described in Chapter 3. One test set consists of identifier names selected at random from the database. Five test sets consist of random selections of particular species of identifier name. The seventh set consists of identifier names composed of a single case only (see Table 4.1).

I manually tokenised each test set of 4,000 identifier names to provide reference sets of tokenisations. The resulting text files consist of lines composed of the identifier name followed by a tab character and the tokenised form of the identifier name, normalised in lower case, with each token separated by a dash, e.g. `HTMLEditorKit` ⟨*tab*⟩ `html-editor-kit`. Bias may have been introduced to the experiment by the reference tokenisations having not been created independently and I discuss the implications below in Section 4.4.4 Threats to Validity.

The identifier names in the test sets were classified using four largely mutually exclusive categories that reflect particular features of identifier name composition related to the difficulty of accurate tokenisation. The categories are:

- **Conventional** identifier names are composed of groups of letters divided by internal capitalisation (lower case to upper case boundary) or separator characters.

- **Digits** identifier names contain one or more digits.

- **Single case** identifier names are composed only of letters of the same case, or begin with a single upper case letter with the remaining characters all lower case.

- **UCLC** identifier names contain two or more contiguous upper case characters followed by a lower case character.

Table 4.1: Distribution of identifier name categories in datasets

Dataset	Description	Conventional	Digits	Single Case	UCLC
A	Random identifier names	2414	467	1011	106
B	Class names	3133	185	113	569
C	Method names	3459	116	184	151
D	Field names	2717	401	818	64
E	Formal arguments	2754	250	961	34
F	Local variable names	2596	349	1021	34
G	Single case	0	0	4000	0

Identifiers names are categorised by first testing for the presence of one or more digits, then testing for the UCLC boundary. Consequently the digits category may contain some identifier names that also have the UCLC boundary. In the seven test sets there are a total of 1768 identifier names containing digits, of which 62 also contain a UCLC boundary. The classification system is intended to allow the exclusion of identifier names containing digits from evaluations of those tools that do not attempt realistic tokenisation of such identifier names, and to allow evaluation of my approach to tokenising identifier names containing digits. The distribution of the four categories of identifier names in each of the datasets is given in Table 4.1.

The 60 projects in the database were surveyed to understand the proportions of each typographical feature in each project, and whether particular features were more likely to occur in a given species of name. Figure 4.1 shows the distribution of each category as a proportion of the total number of unique identifier names in each application, where the whiskers extend at most to 1.5 times the interquartile range from the box. Identifier names containing only conventional boundaries are by far the most common form of identifier name found in all the projects surveyed. A significant proportion of single case identifier names are found in most projects, and around 10% of identifier names contain digits or the UCLC boundary. Table 4.2 gives a breakdown of the proportion of unique identifier names in each category across all 60 projects for each species of identifier. Test sets B to F reflect the most common species, with the exception of constructor names which are lexically identical to class

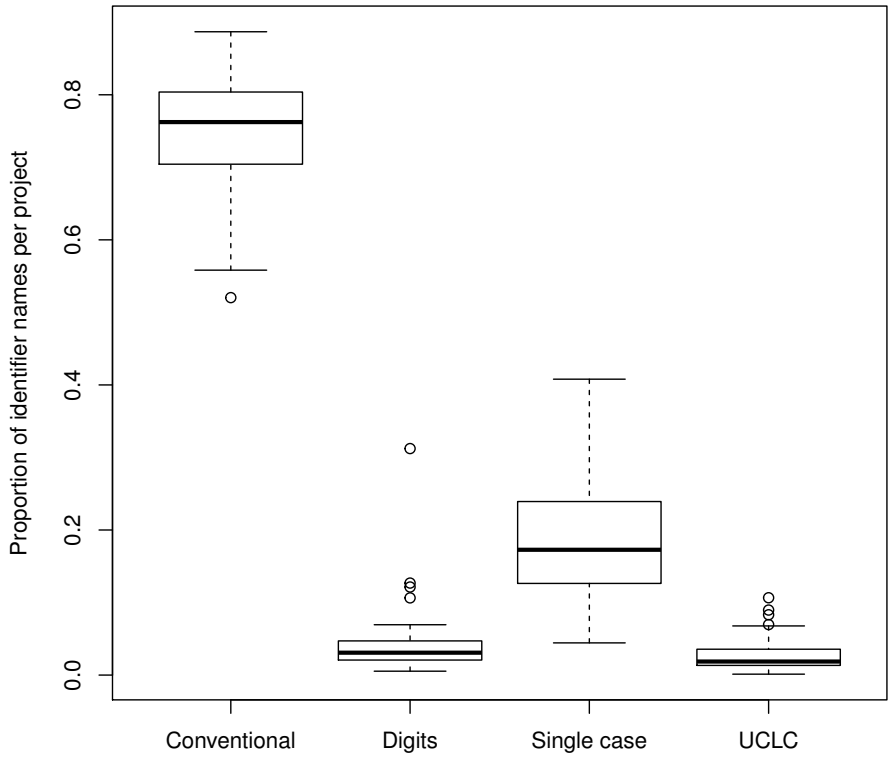


Figure 4.1: Distribution of the percentage of unique identifier names found in each category for sixty Java projects

identifier names, but differ in distribution because not all classes have an explicitly declared constructor, while others have more than one.

Table 4.2 shows that identifier names containing digits and those containing UCLC boundaries constitute nearly 9% of all the identifier names surveyed. Class, constructor and interface identifier names, the most important names for high level global program comprehension, have a relatively high incidence of identifier names containing the UCLC boundary: 13% for class and constructor identifier names and 32% for interface identifier names. In other words, approximately 20% of class names and 40% of interface names require more sophisticated heuristics to determine how to tokenise them.

I evaluated the performance of `INTT` by assessing the accuracy with which the test sets of identifier names were tokenised, and by comparing `INTT` with an implementation of the Samurai algorithm, both in terms of accuracy and the relative strengths and weaknesses of the two approaches.

Table 4.2: Percentage distribution of identifier name categories by species

Species	Conventional	Digits	Single case	UCLC	Overall %
Annotation	70.4	0.2	25.6	3.8	**0.1**
Annotation member	49.8	0.5	49.5	0.2	**<0.1**
Class	79.8	4.1	2.9	13.2	**9.8**
Constructor	79.8	3.5	3.1	13.5	**7.2**
Enum	73.4	0.5	19.4	6.7	**0.1**
Enum constant	55.9	10.2	33.6	0.2	**0.8**
Field	86.1	6.0	6.2	1.7	**27.1**
Formal argument	81.8	3.0	14.2	0.1	**8.1**
Interface	59.3	2.6	6.4	31.7	**1.5**
Label name	59.1	15.7	25.0	0.1	**0.1**
Local variable	82.4	3.8	12.6	1.2	**16.9**
Method	91.6	2.9	1.6	3.9	**28.4**
Total	**84.9**	**4.1**	**6.4**	**4.6**	

4.4.1 `INTT`

I used `INTT` to tokenise the identifier names in each of the seven datasets. The accuracy of the tokenisations was automatically checked against the reference tokenisations for each dataset using a small Java program. A percentage accuracy score calculated for `INTT`'s overall performance and for each species of identifier name. A percentage accuracy was also calculated for each of the four structural categories found in each set of identifier names, see Table 4.3. (The results for dataset G are reported in Section 4.3.4.)

`INTT` was found to have an overall accuracy of 96-97%, which improves marginally when identifier names containing digits are excluded. Identifier names containing digits are tokenised with an accuracy in excess of 85% for three of the six data sets A–F. However, accuracy drops to 64% for method identifier names containing digits. Inspection of the tokenisations for class and method names show that there are two contributing factors: firstly, the assumption that a recognised acronym containing digits always takes precedence over the heuristics when determining a tokenisation led to incorrect tokenisations in some instances and, secondly, some oversplitting of textual tokens occurs. An example of the former is the method name `replaceXpp3DOM` (NetBeans) which was tokenised as {`replace`, `Xpp`, `3D`, `OM`}

Table 4.3: Percentage accuracies for `INTT`

Dataset		Conventional	Digits	Single case	UCLC	Overall	Without digits
A	Random identifier names	97.3	95.9	97.4	85.8	**96.9**	**97.0**
B	Class names	98.3	85.4	92.4	92.1	**96.5**	**97.1**
C	Method names	97.1	63.8	96.8	92.7	**96.0**	**96.9**
D	Field names	97.5	88.7	96.4	87.5	**96.3**	**97.1**
E	Formal arguments	98.8	94.4	93.4	79.4	**97.0**	**97.2**
F	Local variable names	98.2	94.3	92.0	85.3	**96.2**	**96.3**

on the basis that `3D` is a known acronym containing digits. Applying the heuristics alone, however, would have found the correct tokenisation of {`replace`, `Xpp3`, `DOM`}.

The overall percentage accuracy for each dataset is comparable with the accuracies reported for the Samurai tool (Enslen *et al.*, 2009) (97%) and by Madani *et al.* (2010) (93-96%). The breakdowns for each structural type of identifier name show that `INTT` performs less consistently for identifier names containing digits and for those containing the UCLC boundary.

4.4.2 Comparison With Samurai

To make a comparison with the work of Enslen *et al.* I developed an implementation of the Samurai tool based on the published pseudocode and textual descriptions of the algorithm (Enslen *et al.*, 2009). The implementation processed the seven test sets of identifier names and the resulting tokenisations were scored for accuracy against the reference tokenisations. The results are shown in Table 4.4 with the exception of the single case dataset G, which is reported below in Section 4.4.3. The overall accuracy figure given for my implementation of the Samurai algorithm in Table 4.4 excludes identifier names with digits, and should be compared with the figures in the rightmost column of Table 4.3. Samurai's treatment of digits as discrete tokens leads to an accuracy of 80% or more for all but class and method identifier names, where accuracy falls to 45% and 55% respectively.

My implementation of the Samurai algorithm performs less well than the original (Enslen *et al.*, 2009). On inspecting the tokenisations I found more oversplitting than anticipated.

Table 4.4: Percentage accuracies for Samurai

Dataset		Conventional	Digits	Single case	UCLC	Without digits
A	Random identifier names	93.3	92.9	69.1	82.1	**86.3**
B	Class names	94.0	44.9	86.3	81.5	**91.7**
C	Method names	92.8	55.2	88.8	83.4	**92.3**
D	Field names	91.3	78.8	78.2	73.4	**87.7**
E	Formal arguments	94.8	88.4	75.0	64.7	**89.4**
F	Local variable names	92.7	86.2	67.7	70.6	**85.4**

There are a number of factors that could contribute to the observed difference in performance, which are discussed in Section 4.4.4 Threats to Validity.

4.4.3 Single Case Identifier Names

Both `INTT` and Samurai contain algorithms for tokenising single case identifier names that are intended to improve on Feild *et al.*'s greedy algorithm. To compare the two tools I extracted a data set of 4,000 random single case identifier names from the `INVocD` database. All the identifier names consist of a minimum of eight characters: 2,497 are composed of more than one word or abbreviation, the remainder are either single words found in the dictionary or have no obvious tokenisation.

I implemented the greedy algorithm developed by Feild *et al.* following their published description (Feild *et al.*, 2006), to provide a baseline of performance from which the improvement in performance represented by `INTT` and Samurai could be evaluated. The supporting dictionary for the Feild *et al.*'s greedy algorithm was constructed from the English word lists provided with ispell v3.1.20, the same version used by Feild *et al.*. Their stop-list and list of abbreviations were replaced with the same list of abbreviations used in `INTT` and the additional list of terms that are included in `INTT`'s dictionary.

Enslen *et al.* found that Samurai and greedy both had their strengths. Samurai is a conservative algorithm that tokenises identifier names only when the tokenisation is a very much better option than not tokenising. As a result, the greedy algorithm correctly tokenised

identifier names that Samurai left intact. However, as expected, the greedy algorithm was more prone to oversplitting than the more conservative Samurai (Enslen *et al.*, 2009).

The 4,000 single case identifier names were tokenised with 78.4% accuracy by my implementation of Feild *et al.*'s 'greedy' algorithm, with 70.4% accuracy by my implementation of Samurai, and with 81.6% accuracy by `INTT`.

4.4.4 Threats to Validity

The threats to validity are concerned with construct validity and external validity. Internal validity is not considered because I make no claims of causality. Similarly, statistical conclusion validity is not considered, because I have not used any statistical tests.

Construct Validity

There are two key concerns regarding construct validity: the possibility of bias being introduced through manual tokenisation of identifier names used to create sets of reference tokenisations; and the observed difference in performance between my implementation of Samurai and the accuracy reported for the original implementation (Enslen *et al.*, 2009).

That I split the identifier names for the reference tokenisations may have introduced a bias towards tokenisations that favour `INTT`. I guarded against this during the manual tokenisation process as much as possible, and conducted a review of the reference sets to look for any possible bias and revised any such tokenisations found. Of the related works (Feild *et al.*, 2006; Enslen *et al.*, 2009; Madani *et al.*, 2010) only Enslen *et al.* used an independently created set of reference tokenisations.

I have identified three factors that may explain the reduced accuracy achieved by my implementation of Samurai in comparison to the reported accuracy of the original. When implementing the Samurai algorithm, I took all reasonable steps, including extensive unit testing, to ensure the implementation conformed to the published pseudo code and text descriptions (Enslen *et al.*, 2009). However, it is possible that I may have inadvertently introduced errors. There is the possibility that computational steps may have inadvertently been omitted from the published pseudo code description. The third possibility is that the scoring formula used in Samurai to identify preferable tokenisations, which was derived empirically, may not hold for oracles composed of fewer tokens with lower frequencies. The

oracle used in my implementation of Samurai was constructed using identifier names found in 60 Java projects, much fewer than the 9,000 projects Enslen *et al.* used as the basis for their dictionary. My version of the Samurai oracle contains 61,580 tokens, with a total frequency of 3 million. In comparison the original Samurai oracle was created using 630,000 tokens with a total frequency of 938 million.

External Validity

External validity is concerned with generalisations that may be drawn from the results. My experiments were conducted using identifier names extracted from Java source code only. Although I cannot claim any accuracy values for other programming languages, I expect results to be similar for programming languages with similar naming conventions, because the tokenisation approach is independent of the programming language. The experiments were also conducted on identifier names constructed using the English language. While the techniques and the tool I developed can be applied readily to identifier names in other natural languages, some of the heuristics, in particular the treatment of '2' and '4' as homophone substitutions for prepositions, may need to be revised for natural languages other than English.

A further concern for external validity is that I followed the experimental design of Enslen *et al.* (2009). The algorithms implemented in `INTT` are intended to address the challenges posed to accurate tokenisation by specific typographical features. With the exception of single case identifier names, identifier names for the test sets were selected at random from each species rather than by typographical feature. A better evaluation of the algorithms might have been achieved by extracting sets of 4,000 identifier names containing specific typographical features such as digits or the UCLC boundary. However, it should be noted that randomly selected sets of names were also used in the later evaluation of multiple tokenisers by Hill *et al.* (2013).

4.5 Discussion

A motivation for adopting the approach described above was a concern over the computing resources, both in terms of time and space, that were being devoted to solving the problem of identifier name tokenisation. The approach taken by Guerrouj *et al.* processes each identifier name in detail and is thus relatively computationally intensive, while the Samurai algorithm

relies on harvesting identifier names from a large body of existing source code — a total of 9,000 projects — to create the supporting oracle. Like Samurai, `INTT` processes identifier names selectively and reserves more detailed processing for those identifier names assumed to be more problematic. However, accuracy levels similar to the published figures for Samurai are achieved — both as part of my evaluation and the independent evaluation conducted by Hill *et al.* (2013) — using a smaller oracle constructed, largely, from readily available components such as the SCOWL word lists.

4.5.1 Identifier Names Containing Digits

I have demonstrated an approach to tokenising identifier names containing digits that achieves an accuracy of 64% at worst and most commonly 85%–95%. The only tool available for comparison was my implementation of the Samurai algorithm, which takes a simple and unambiguous approach to tokenising identifier names containing digits and achieves, an accuracy that is consistently between 10% and 3% less than that achieved by `INTT`, with the exception of class identifier names where Samurai's treatment of digits as discrete tokens results in an accuracy of 45%, some 40% less than `INTT`.

While I am largely satisfied with having achieved such high rates of accuracy, there is room for improvement. Inspection of `INTT`'s output showed that some inaccurate tokenisations could be attributed to incorrect tokenisation of textual portions of the identifier name. However, they also showed that some of my heuristics for identifying how to tokenise around digits require refinement. One possibility is the introduction of a specific heuristic for tokens of the form 'v5', signifying a version number, so that they are tokenised consistently. I found that though most were tokenised accurately, some identifier names, for example `SPARCV9FlushwInstruction` (JDK), were not. The difficulty appears not to be the digit alone, but that the digit in combination with the letter is key to accurate tokenisation. Other incorrect tokenisations occurred where identifier names such as `replaceXpp3DOM` contain a known acronym. The solution in such cases appears to be to choose between the tokenisation resulting from using recognised acronyms, and that arising from the application of the heuristics alone.

Another solution that may be used with certain types of digit-containing acronym is to employ pattern matching. For example developers will, very occasionally, refer to ISO

and RFC standards in names and these could easily be matched with regular expressions. Alternatively, additional heuristics could be added to the list in Section 4.3.3 to recognise established acronym forms.

4.5.2 Limitations

No current approach tokenises all identifier names accurately. Indeed, accurate tokenisation of all identifier names may only be possible with some projects where a given set of identifier naming conventions are strictly followed. However, there are a number of barriers to tokenisation that are difficult to overcome, and outside the control of those processing source code to extract information. An underlying assumption of the approaches taken to identifier name tokenisation is that identifier names contain semantic information in the form of words, abbreviations and acronyms and that these can be identified and recovered. Developers, however, do not always follow identifier naming conventions and building software that can process all the forms of identifier names that developers can dream up is most likely impossible and would require a great deal of additional effort for a minimal increase in accuracy. For example, `is0x8000000000000000L` (Xerces) is an extremely unusual form of identifier name — the form is seen only three times[5] in the 60 projects surveyed — which would require additional functionality to parse the hexadecimal number in order to tokenise the identifier name accurately. However, the approach suggested above to use a form of pattern matching to identify certain types of digit-containing acronym might be extended to include hexadecimal numbers. The disadvantage is that such a technique introduces additional computational overhead for what are very rare cases.

Another limitation arises from the use of neologisms and misspelt words. Neologisms found in the single case test set include 'devoidify', 'detokenated', 'grandcestor', 'indentator', 'pathinate' and 'precisify'. With the exception of 'grandcestor' these are all formed by the unconventional use of prefixes and suffixes with recognised words or morphological stems. Some, e.g. 'pathinate', are vulnerable to oversplitting by Feild *et al.*'s greedy algorithm and algorithms based on it. Others may cause problems when concatenated with other words in single case identifier names where a plausible tokenisation is found to span the intended boundary between words.

[5] NetBeans' unit tests include the method names `test0x01` and `test0x16`.

Samurai and `INTT` both guard against oversplitting neologisms by using lists of prefixes and suffixes. `INTT` identifies single case identifier names found to be formed by a recognised word in combination with either or both a known prefix or suffix and does not attempt to tokenise them. Samurai tries to tokenise all single case identifier names, but rejects possible tokenisations where one of the resulting tokens would be a known prefix or suffix. All of the neologisms listed are recognised as single words by both approaches. However, `INTT` does not recognise 'precisify' as a neologism resulting from concatenation and would try to tokenise it. The use of stemming may refine the recognition of neologisms, but would also add computational overhead. The recognition of word blends such as 'grandcestor' is non-trivial (Cook and Stevenson, 2010).

Tools that use natural language dictionaries as oracles will try to tokenise a misspelt word, whether it is found in isolation or concatenated with another word, as a single case identifier name. The majority of observed misspellings appear to be simple and result from insertion of an additional letter, omission of a letter or transposition of two letters. Precisely the sort of problem that can be readily identified by a spell checker. For example, `possition` (NetBeans) is oversplit by both `INTT` and the greedy algorithm as {`pos`, `sit`, `ion`} and {`poss`, `it`, `ion`}, respectively. My implementation of Samurai also oversplits `possition` probably because of a combination of the relative rarity of the spelling mistake, and the more common occurrence of the token `poss` (AspectJ, Eclipse, Netbeans, and Xalan). A step towards preventing some oversplitting of misspelt words could be achieved through the use of algorithms applied in spell-checking software, such as the Levenshtein distance (Levenshtein, 1966), or using a spellchecker. Since this work was completed, I have adapted spellchecking software for use with identifier names (See Chapter 6) and am examining ways of incorporating it into `INTT`.

Inspection of the tokenisations of the test sets for each tool show that Feild *et al.*'s greedy algorithm is prone to oversplitting neologisms particularly where a suffix such as 'able' that is also a word has been added to a dictionary word, e.g. `zoomable` (JFreeChart). Feild *et al.*'s greedy also cannot consistently tokenise identifier names that start and end with abbreviations not found in its dictionary, e.g. `tstampff` (BORG Calendar), and cannot differentiate between ambiguous tokenisations. Indeed, Feild *et al.* provide no description of how to differentiate between tokenisations that return identical scores (Feild *et al.*, 2006). In my implementation of the greedy algorithm, the tokenisation resulting from the backward

pass is selected in such situations, because English language inflections, particularly the single 's', can be included by the forward pass of the algorithm. For example, `debugstackmap` (JDK) is tokenised incorrectly as {`debugs`, `tack`, `map`} by the forward pass and correctly as {`debug`, `stack`, `map`} by the backward pass. The backward pass is also prone to incorrect tokenisations, though from inspection of the test set this is much less common. For example, the reverse pass tokenises `commonkeys` (JDK) as {`com`, `monkeys`}, using ispell word lists where 'com' is listed as a word.

4.5.3 Future Work

Tools such as `INTT` and Samurai work on the assumption that developers generally follow identifier naming conventions and that additional computational effort is required for exceptions that can be identified. As noted in the description of the problem (Section 4.1) the assumption is an approximation. There are many cases where the typographical conventions on word division are broken, or are used in ways that divide the elements of semantic units so as to render them meaningless. In other words, a key issue for tokenisation tools is that word divisions, be they separator characters or internal capitalisation, can be misleading and are thus not always reliable. Consequently, meaningful tokens may need to be reconstructed in client tools by concatenating adjacent tokens. An alternative would be for the tokeniser to reconstruct tokens, which may be relatively straightforward for some of the observed names such as `subMenu`. However, more complex instances, may require other approaches, such as heuristics that rely on the expected grammatical form of the name. In the longer term, more computationally intensive processes, exemplified by Normalize (Lawrie and Binkley, 2011) and LINSEN (Corazza *et al.*, 2012), can be refined to provide more accurate tokenisers.

4.6 Summary

Identifier names are the main vehicle for semantic information during program comprehension. The majority of identifier names consist of two or more words, abbreviations or acronyms concatenated and therefore need to be tokenised to recover their semantic constituents, which can then be used for tool-supported program comprehension tasks, including concept location and requirements traceability. Tool-supported program comprehension is important for the maintenance of large software projects where cross-cutting concerns mean that concepts are

often diffused through the source code.

While typographical conventions should make the tokenisation of identifier names a straightforward task, they are not always clear, particularly with regard to digits, and developers do not always follow conventions rigorously, either using potentially ambiguous word division markers or none at all. Thus accurate identifier name tokenisation is a challenging task.

In particular, the tokenisation of identifier names of a single case is non-trivial and there are known limitations to existing methods, while identifier names containing digits have been largely ignored by published methods of identifier name tokenisation. However, these two forms of identifier name occur with a frequency of 9% in my survey of identifier names extracted from 16.5 MSLOC of Java source code, demonstrating the need to improve methods of tokenisation.

In this chapter I addressed two research questions concerning how to improve the tokenisation of names with ambiguous or no word boundaries (RQ 2), and how to tokenise names containing digits (RQ 3). By answering the questions I make two contributions. First, I demonstrate an improvement on current methods for tokenising single case identifier names, on the one hand in terms of improved accuracy and scope by tokenising forms of identifier name that current tools cannot, and on the other hand in terms of resource usage by achieving similar or better accuracy using an oracle with less than 20% of the entries used by Samurai. Furthermore, the oracle used can be constructed easily from available components, whereas the Samurai algorithm relies on identifier names harvested from 9,000 Java projects. Second, I have introduced an original method for tokenising identifier names containing digits that can achieve accuracies in excess of 90% and is a consistent improvement over a naive tokenisation scheme.

I also make two technical contributions. Firstly, `INTT`, written in Java, is available for download[6] as a JAR file with an API that allows the identifier name tokenisation functionality described in this chapter to be integrated into other tools. Secondly, the data used to test `INTT` is made available as plain text files. The data consists of the seven test datasets of 28,000 identifier names together with the manually obtained reference tokenisations, and 1.4 million records of over 800,000 unique identifier names in 60 open source Java projects,

[6] `http://oro.open.ac.uk/28352/`

including information on the identifier species. By making these computational and data resources available, I hope to contribute to the further development of identifier name based techniques (not just tokenisation) that help improve software maintenance tasks.

Having developed a means of extracting and storing identifier names for analysis, and devised and implemented improved techniques for the tokenisation of names, I now have the fundamental tools needed to support the analysis of names. In the following chapter I describe analysis of the structure of Java class names.

Chapter 5

The Analysis of Class Identifier Names

The preceding two chapters have focused on the development and implementation of a system to extract identifier names from source code and to store the names with their tokens, and on the tokenisation of names. In the remaining chapters of the dissertation the focus returns to the principal research question: *What types of content and phrasal structure do developers use in Java class and reference names?* The class and reference names, and their tokens identified by `INTT`, extracted from 60 Java projects and stored in the `INVocD` database are the data source for the analysis reported in this and the subsequent two chapters. This chapter reports on an investigation of the class names in the corpus. The following two chapters report investigations of reference names: Chapter 6 describes a survey of reference names, and Chapter 7 investigates the adherence of reference names to naming conventions.

Class identifier names represent the core entities and concepts encoded in object-oriented source code, and are vital to program comprehension (Rajlich and Wilde, 2002; Deißenböck and Pizka, 2006). Naming conventions advise that developers choose 'meaningful' identifier names and that class identifier names should be nouns (Vermeulen *et al.*, 2000; Gosling *et al.*, 2014) or descriptive nouns (Gosling *et al.*, 2014). More advanced practitioner texts advocate a considered approach to identifier naming (Beck, 2008), and a variety of conventions of language use have arisen as a result of praxis (Liblit *et al.*, 2006).

Analysis of C function identifier names by Caprile and Tonella (1999) led to support for automated name refactoring (Caprile and Tonella, 2000). Similar analysis of Java method

identifier names found a link between identifier names and method implementation (Høst and Østvold, 2008), which was successfully leveraged to identify candidates for name refactoring and to suggest possible refactorings (Høst and Østvold, 2009).

Despite the importance of class identifier names for program comprehension, there is no detailed understanding of their structure. In this chapter I present a survey of the class identifier names found in 16.5 MSLOC of open source Java projects and recorded in `INVocD` (See Chapter 3). In particular I seek to answer the following two research questions that address different aspects of the principal research question:

RQ 4 What phrasal structures do developers use in class names?

RQ 5 How do developers incorporate super class and interface names in class names?

I adopt two approaches to the analysis of class names. The first, used to answer both RQ 4 and RQ 5, analyses the identifier names to recover patterns of parts of speech used in their construction and identify common phrasal structures with the intention of determining the extent of the use of the naming convention that a class name is a noun or noun phrase and what alternatives are used in practice. The second approach, focusing on RQ 5, catalogues the repetition of the component words of super class and interface (super type) names in class identifier names. Such reuse is evident in the Java library, for example, where the `java.util` classes `HashSet` and `TreeSet` retain the name of the `Set` interface implemented by their common super class `AbstractSet`. However, little is known of the extent to which these types of pattern are replicated in production source code and under what circumstances.

5.1 Related Work

Knowledge of the structure of identifier names has practical applications in source code comprehension and software development and maintenance. Analysis of C function names by Caprile and Tonella (1999) has been applied to automate the refactoring of names (Caprile and Tonella, 2000). Høst and Østvold undertook detailed analysis of Java method names using a specially developed PoS tagger (Høst and Østvold, 2008) and found relationships between method names and method implementation (Høst and Østvold, 2009) in terms of the

micro-patterns, particular characteristics and operations, found in the compiled bytecode. For example, a mutator (setter) method might have no return type, a single argument, and assign that argument to a class field. This knowledge was then applied to develop the automated detection of method naming errors and recommendation of candidate refactorings (Høst and Østvold, 2009). As well as identifying conventional parts of speech, Høst and Østvold's PoS tagger treats type names as a separate part of speech.

Singer and Kirkham (2008) identified a link between Java class names and the micro-patterns (Gil and Maman, 2005) found in the implementation of a class using the approximation that Java class names are of the form $\mathsf{JJ}^*\mathsf{NN}^+$, where JJ represents an adjective and NN a noun[1]. However, the link was based on the assumption that the rightmost noun is an indicator of the class's implementation and is the basis of the relationship with the micropatterns found in the class implementation, and no detailed analysis of the structure and content of class identifier names was undertaken.

The structure of Java class identifier names was investigated as part of a study of the cognitive aspects of identifier names as a form of communication (Liblit *et al.*, 2006). The investigation found developers used a variety of morphological and grammatical artifices when constructing identifier names, many of which are not proposed by naming conventions. However, the investigation was conducted using source code from multiple programming languages, not Java alone, and was not a comprehensive survey of naming practice. The work of Liblit *et al.* influenced the pragmatic grammar for extracting knowledge from identifier names found in the Software Word Usage Model (SWUM) developed by Hill (2010), and the system of template sentences developed by Abebe and Tonella (2010).

SWUM supports feature location and program comprehension by constructing multi-layered models, including light ontologies, that combine the structural information in source code and semantic information extracted through natural language analysis of identifier names. As part of SWUM, Hill (2010) developed a pragmatic grammar that describes a wide range of phrasal content expected to be found in identifier names to support the extraction of information. The grammar is based in naming conventions, and Liblit *et al.*'s and Hill's own observations.

Identifier naming conventions were used by Abebe *et al.* (2009) to identify *smells* in source

[1]The Penn Treebank PoS tag set is used to annotate parts of speech `ftp://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz` a summary of which is given in Appendix E

code. The smells are predicated on deviations from suggested identifier naming conventions that arise from programming conventions, and, to a lesser extent, deviation from established conventions arising from identifier naming praxis. A single rule concerns the grammatical structure of class identifier names, and states that class identifier names should contain at least one noun and not contain any verbs. However, this work is based on naming conventions that are guidelines of good practice, rather than a detailed knowledge of naming conventions found in practice. Abebe and Tonella (2010) used an approximation of the phrasal structure of identifier names to develop a system where an off the shelf PoS tagger and parser, Minipar, was used to tag the name as part of a 'template' sentence. At the simplest, the class name is tokenised and the phrase 'is a thing' added to create a simple sentence. Abebe and Tonella used four templates for class names: two for names beginning with nouns, and two for those beginning with verbs. A key idea for this investigation is that class identifier names found in practice begin with verbs, something that is not specified in naming conventions or in the work of Liblit *et al.* (2006). Abebe and Tonella's work concerns the development of a system for parsing identifier names and they do not comment on the proportion of class names they are able to parse with this technique.

Other techniques have been applied to extract ontologies from source code. Raţiu (2009) and Falleri *et al.* (2010) have reverse engineered ontologies from source code using identifier names a source of information. Raţiu constructed ontologies to investigate the logical structure of domain knowledge encoded in source code. Falleri *et al.* created WordNet (Fellbaum, 1998) style semantic nets from identifier names. To determine the meaning of an identifier name, Falleri *et al.* used the TreeTagger PoS tagger[2] and applied a version of dependency analysis, a technique for analysing sentences, to Java identifier names to identify the *dominant* component words and, thereby, the meaning.

Deißenböck and Pizka (2006) proposed a scheme of concise and consistent naming where a single identifier name represents a single concept throughout the program, and that identifier names are composed so as to represent discrete concepts unambiguously. They found that application of their naming approach improved the maintainability of an in-house project. In a follow up experiment, Lawrie *et al.* (2007a) surveyed the identifier names found in 48 MSLOC of C, C++, Fortran and Java source code to determine the extent of violations of concise

[2] `http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/`

and consistent naming. The syntactic methodology employed by Lawrie *et al.* identifies potential violations of concise and consistent naming which include some conventional patterns of naming found in Java inheritance trees.

Since the completion of this work, Gupta *et al.* (2013) have created an identifier name specific PoS tagger. The tagger uses Hill's grammar as an approximation of identifier names found in practice. All possible PoS tags for the tokens of an identifier name are identified using WordNet (Fellbaum, 1998), and a likely tagging for the name accepted on the basis of word frequency lists and the grammar. The focus of Gupta *et al.*'s work is on the development of the PoS tagger and evaluation of its performance, and does not provide an insight into the detailed structure of class names.

Previous research has investigated identifier naming in Java and detailed investigations have been constrained to method identifier names. Approximations of class name structure have been used to support the extraction of semantic information from names. However, despite the practical applications for a comprehensive understanding of the structure of identifier names, there has been no detailed investigation of Java class identifier names.

5.2 Methodology

The corpus of class names used in the study consists of the 120,000 unique class identifier names recorded in `INVocD`. Two separate analytical techniques were applied to each identifier name: PoS tagging to help identify any common grammatical patterns, and an investigation of the origins of class identifier name components, if any, found within the names of the immediate super class and implemented interfaces. I also undertook a case study of FreeMind[3], a Java mind-mapping application that is part of the corpus, to investigate whether unconventionally structured class identifier names indicate that either there is a problem with the class name — so that it can be refactored to a more conventionally structured name — or that there is a problem with the class itself. And, in the latter case, whether any possible refactorings could be identified that might result in two or more conventionally named classes.

[3]FreeMind v0.9.0RC9 `http://freemind.sourceforge.net/`

5.2.1 Analysis of Grammatical Composition

Previous investigations of C function names and Java method names (Caprile and Tonella, 1999; Høst and Østvold, 2008; Falleri *et al.*, 2010) used PoS tagging to identify grammatical patterns in names. Some teams have created their own PoS tagger developed for identifier names (Høst and Østvold, 2008; Gupta *et al.*, 2013), while others have used an off-the-shelf PoS tagger (Falleri *et al.*, 2010; Abebe and Tonella, 2010; Hill, 2010; Binkley *et al.*, 2011). I followed the latter route by using the Stanford Log-linear PoS tagger[4].

My initial experiments were undertaken with the default tagger provided with the Stanford PoS tagger. Using the tagger, which is trained on a corpus of articles taken from The Wall Street Journal, I observed, through manual inspection, error rates of 15–30% for whole identifier names, depending on the project analysed. The chief sources of error appeared to be the difference between the structure of Java class names and the conventional English sentences that form the tagger's training corpus, and the presence of abbreviations and a technical vocabulary. The consequence of this was that the tagger was trying to tag unknown words in an unrecognised context. As a result common English words were tagged as foreign words. I also observed issues related to the resolution of ambiguous PoS. For example the class name `ContentHandler` consisting of two nouns was consistently tagged as an adjective followed by a noun, and the word 'set' was often tagged as a verb when used as a noun.

I experimented with 'templates' similar to those used by Abebe and Tonella (2010) by creating more sentence-like structures from class names by including the Java keywords from the class declaration and appending the name of the super class and implemented interface. For example, for the class `CustomPropertiesTagHandler`, found in GanttProject, which has no explicit super class and implements two interfaces, I constructed the phrase "*class custom properties tag handler implements tag handler and parsing listener*". There was no significant improvement in PoS tagging accuracy.

I decided to train a PoS tagger model using the Stanford PoS tagger. A training corpus of 9,000 class names was extracted at random from 13 of the 60 projects analysed, including the Java library. I split and tagged each name manually and any class names that could not be unambiguously tagged were discarded. A tagger model was trained using the tagged class names. A separate test corpus of 2,000 class identifier names was created by manually

[4] `http://nlp.stanford.edu/software/tagger.shtml`

tagging class names extracted randomly from a further 8 of the projects analysed. An accuracy of 95% was achieved when tagging individual words and abbreviations against the test corpus. However, unrecognised words and abbreviations were only tagged with 83% accuracy, resulting in an accuracy rate for whole identifier names of 87% being reported by the tagger in test mode.

Whilst not perfect, the accuracy is an improvement on the default tagger model, and manual inspection appeared to show less variability. Høst and Østvold (2008) state the accuracy of their PoS tagger is better than 97% as the result of manual inspection, but are unclear whether this figure is for individual words or for whole method identifier names. Falleri *et al.* (2010) claim a PoS tagging accuracy of 96% for TreeTagger's default tagger for identifier names, but, again, are unclear whether this relates to individual tags, or whole identifier names.

5.2.2 Analysis of Inheritance

The common grammatical structures found in class identifier names do not identify how developers encode information in class names. Class names in some inheritance hierarchies in the Java library repeat part of the super class name. For example the class `HTMLEditorKit`, found in the `javax.swing.text.html` package, is a subclass of `StyledEditorKit`. Similarly the collections classes in the `java.util` package often follow a pattern of naming where a base interface name is retained through intermediate classes to the various implementations. Taking the `List` classes as an example, the `List` interface extends the `Collection` interface, and is implemented in a class by `AbstractList`. The common list classes — e.g. `ArrayList` and `LinkedList` — then extend `AbstractList`, i.e. specialised implementations extend the abstract class and replace `Abstract` with an adjective or adjectival phrase describing the implementation.

I analyse the incorporation of component words in a class identifier name from the immediate super class and any implemented interfaces. Class names were partitioned into six groups according to whether the class explicitly extends a super class — Java classes that do not explicitly declare a super class extend the root class `Object` — and the number of interfaces implemented. The notation I developed consists of the letters E for extends and I for implements with each letter being followed by a subscript indicating the number of

super classes extended or interfaces implemented: the values of the subscript being 0, 1 and n where the last means two or more and can only be a subscript to I. For example a class that extends a super class and implements no interfaces is classified as $E_1 I_0$.

I investigate lexical inheritance, that is I investigate whether component words from the super class or implemented interface names are found in the class identifier name. Accordingly no distinction is drawn between identically named super classes from different packages and the analysis ignores any package name that might have been specified by the developers in the extends and implements clauses of class declarations. Similarly generic type names are ignored where they are specified.

5.2.3 Case Study

FreeMind is a mind-mapping application written in Java with an 11 year development history. The application consists of a GUI that allows the user to edit, format and annotate a treelike-graph structure of text nodes. The mind maps are stored in an XML format, and can be exported to a range of external formats including HTML and OpenOffice Writer, and as images, flash animations and Java applets. I selected FreeMind as the subject for the case study because of its maturity, its relatively modest size (some 650 classes and 41 KSLOC). I have also used FreeMind for more than 5 years and am familiar with its functionality.

Using the results of the grammatical and the inheritance analyses, I identified those classes with identifier names that do not conform to the commonly occurring grammatical patterns found in FreeMind class names. The subset of classes was then inspected to determine whether the class might be named according to one of the more commonly occurring grammatical patterns, or, if the class appeared to be appropriately named, whether the unusual name might be indicative of a potential refactoring of the class into two or more classes with more conventional names.

5.3 Results

To present the results, individual Treebank adjective, noun and verb PoS tags are combined so that JJ, NN and VB include all forms of adjective, noun, and verb respectively. By collapsing the Penn Treebank categories a simplified tagset was created, similar to that used by Høst and Østvold (2009).

5.3.1 Grammatical Structure (RQ 4)

Table 5.1 shows the absolute and relative frequencies for the most common grammatical patterns found in 120,000 class identifier names extracted from 60 open source Java projects and recorded in `INVocD`. The four grammatical forms given in the table comprise 90% of the identifier names analysed. The remaining 10% of class identifier names are formed using a variety of patterns, some with only a single instance. Of note is that Singer and Kirkham's approximation of Java class identifier names, $\mathsf{JJ}^*\mathsf{NN}^+$ (Singer and Kirkham, 2008), which reflects the common convention that class names should be nouns (Gosling *et al.*, 2014; Vermeulen *et al.*, 2000; The Eclipse Foundation, 2007), can be realised by merging the two most common categories and includes 85% of the class identifier names analysed. The third most common form, $\mathsf{NN}^+\mathsf{JJ}^+\mathsf{NN}^+$, is also a noun phrase, e.g. `HttpThrowableReporter` (GWT) and `CaseInsensitiveMap` (Tapestry), making 88% of the class names analysed nouns or noun phrases.

Table 5.1: Common Part of Speech Patterns and Frequencies for all projects

Pattern	Absolute Frequency	Relative Frequency
NN^+	88489	0.73
$\mathsf{JJ}^+\mathsf{NN}^+$	14833	0.12
$\mathsf{NN}^+\mathsf{JJ}^+\mathsf{NN}^+$	3579	0.03
$\mathsf{VB}\,\mathsf{NN}^+$	2918	0.02

Around 2% of class names begin with a verb. Inspection of the classes reveals that many represent actions in GUIs,

The answer to RQ 4 is that some 88% of class names are, as expected, and as observed by others (Singer and Kirkham, 2008; Liblit *et al.*, 2006), nouns or noun phrases, and a further 2% have a structure that appears to be a verb phrase. The remaining 10% have structures that range from the simple, e.g. `NotEmpty` (a mixin class in Tapestry) $\mathsf{RB}\,\mathsf{JJ}$, to the more complex, e.g. `MultipleArtifactsNotFoundException` (Maven) $\mathsf{JJ}\,\mathsf{NNS}\,\mathsf{RB}\,\mathsf{VBN}\,\mathsf{NN}$. The latter example has a sentence, rather than a phrase, structure even though a Java developer may well consider it to represent a single concept, and, to all intents and purposes, to be a surrogate noun.

5.3.2 The Influence of Inheritance (RQ 5)

Table 5.2 shows the distribution of classes according to type of inheritance in classes found in all 60 projects. In total 80% of classes are related to another class by inheritance. Some 58% of classes extend a super class, with the majority, 45% in the E_1I_0 category, not implementing an interface. Overall, 35% of classes implement one or more interfaces, and, reflecting the situation with class-based inheritance, the majority of classes implementing an interface do not also extend a class. Indeed only 13% of classes, overall, take advantage of both dimensions of inheritance available in Java.

Table 5.2: Distribution of inheritance categories for all projects

Category	Absolute Frequency	Relative Frequency
E_0I_0	25056	0.21
E_0I_1	22000	0.18
E_0I_n	4280	0.04
E_1I_0	54232	0.45
E_1I_1	11668	0.10
E_1I_n	3350	0.03

Figure 5.1 shows the variation in the proportions of classes in the inheritance categories for the 60 projects investigated, where the whiskers extend at most to 1.5 times the interquartile range from the box. There is considerable variation between projects indicating different teams' preference for the use of particular types of inheritance. For example, E-Gantt and JavaCC have no classes that implement more than one interface, while, at the other end of the scale, 48% of JFreeChart classes do. By far the most marked variation is in the proportion of classes that extend a single super class (E_1I_0) ranging from 70% of classes for ArgoUML to 15% for Tapestry, where 77% of classes are found in the E_0I_0 and E_0I_1 categories.

Table 5.3: Relative frequency of most common grammar patterns by inheritance category

	NN^+	$\mathsf{JJ}^+\mathsf{NN}^+$	$\mathsf{NN}^+\mathsf{JJ}^+\mathsf{NN}^+$	$\mathsf{VB}^+\mathsf{NN}^+$
E_0I_0	0.85	0.08	0.01	0.01
E_0I_1	0.73	0.15	0.02	0.02
E_0I_n	0.75	0.15	0.03	0.01
E_1I_0	0.68	0.12	0.04	0.03
E_1I_1	0.70	0.15	0.04	0.02
E_1I_n	0.75	0.14	0.04	0.02

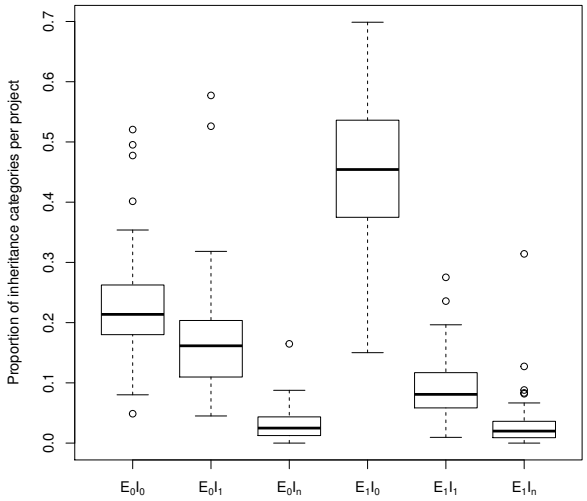


Figure 5.1: Distribution of inheritance categories in 60 Java projects

The distribution of the 4 dominant grammatical patterns in the 6 inheritance categories is shown in Table 5.3. Of note are the similar frequencies of the grammatical forms for the 5 categories where inheritance is involved, and that the E_0I_0 category has a noticeably greater proportion of identifier names composed exclusively of nouns.

Table 5.4 shows the frequency with which elements of super class and interface names are repeated in class identifier names for the different inheritance categories (the E_0I_0 category is omitted from the table). The columns headed *all* contain the frequency for each category with which the super class or interface name is repeated in its entirety and uninterrupted. The columns headed *fragment* give the frequency with which one or more fragments of the super class or interface name are repeated. The *both* column shows the frequency with which elements from both sources are repeated in the class identifier name. For each inheritance category approximately 80% of class identifier names incorporate elements from either the super class or implemented interface identifier names. The repetition of fragments of super class identifier names is more common than the repetition of the entire super class name. Repetition of interface identifier names and fragments occurs with a similar frequency, apart from the E_1I_n category where fragments of interface names are repeated more often than the entire interface name. Where both dimensions of inheritance are used — the E_1I_1 and E_1I_n categories — the super class name is the more common source of class identifier name components. Indeed some 40% of identifier names in the E_1I_1 category and 43% in the E_1I_n category repeat component words from only the super class name, compared to 18% and

22%, respectively, that repeat component words from interface identifier names alone. I also found that 8% of class identifier names in the E_0I_n category and 1% in the E_1I_n category incorporate elements from two or more implemented interface identifier names.

Table 5.4: Relative frequency distribution of name inheritance within inheritance categories for all projects

	Super Class Name		Interface Name		
Category	**All**	**Fragment**	**All**	**Fragment**	**Both**
E_0I_1	-	-	0.39	0.37	-
E_0I_n	-	-	0.38	0.40	-
E_1I_0	0.23	0.58	-	-	-
E_1I_1	0.14	0.53	0.24	0.21	0.27
E_1I_n	0.11	0.50	0.15	0.25	0.18

The most common grammatical forms of class identifier names with component words inherited from a super class or interface are given in Table 5.5 for each category. Table 5.5 introduces a notation for tagging super class and interface names, as well as their fragments. The tags are SC to represent a super class name and SCF for a fragment of the super class name. Similarly, II and IIF are used for interface names and fragments of interface names respectively.

For each category, the most common patterns incorporate the inherited name elements as a suffix. This is the pattern found in the earlier example from the `java.swing.text.html` package where `HTMLEditorKit` is a subclass of `StyledEditorKit`. In other words the part of the super class name that defines what the class is is retained and a word or words are added as a prefix to create the specialised version of the super class. Similarly where part of an interface name is retained as a suffix, the interface name defines the type of thing the class is. For example, the class `GlobPatternMapper` found in ANT implements the `FileNameMapper` interface.

The patterns where a fragment of the super class or interface name is used as a prefix for the class name are less common. An example is the `InstructionHandle` class in MultiJava, which extends the super class `AbstractInstructionAccessor`. In this case the focus of the class's activity, an instruction, is the discriminating term that is perpetuated through inheritance.

Table 5.5: Common grammatical forms of class name component inheritance

Category	Grammatical Form	Relative Frequency
E_0I_1	$\mathsf{NN}^+\mathsf{II}$	0.19
	$\mathsf{NN}^+\mathsf{IIF}^+$	0.17
	$\mathsf{II}\,\mathsf{NN}^+$	0.09
	$\mathsf{IIF}^+\mathsf{NN}^+$	0.05
E_0I_n	$\mathsf{NN}^+\mathsf{II}^+$	0.17
	$\mathsf{NN}^+\mathsf{IIF}^+$	0.15
	$\mathsf{II}^+\mathsf{NN}^+$	0.09
	IIF^+	0.07
	$\mathsf{IIF}^+\mathsf{NN}^+$	0.06
E_1I_0	$\mathsf{NN}^+\mathsf{SCF}^+$	0.30
	$\mathsf{NN}^+\mathsf{SC}$	0.13
	$\mathsf{SCF}^+\mathsf{NN}^+\mathsf{SCF}^+$	0.05
	$\mathsf{SCF}^+\mathsf{NN}^+$	0.05
	$\mathsf{JJ}^+\mathsf{NN}^+\mathsf{SCF}^+$	0.03
	$\mathsf{JJ}^+\mathsf{SC}$	0.03
E_1I_1	$\mathsf{NN}^+\mathsf{SCF}^+$	0.15
	$\mathsf{II}\,\mathsf{SCF}^+$	0.11
	$\mathsf{NN}^+\mathsf{SC}$	0.06
	$\mathsf{SCF}^+\mathsf{IIF}^+$	0.04
	IIF^+	0.04
	$\mathsf{SCF}^+\mathsf{II}$	0.03
	$\mathsf{NN}^+\mathsf{IIF}^+$	0.03
E_1I_n	$\mathsf{NN}^+\mathsf{SCF}^+$	0.20
	$\mathsf{NN}^+\mathsf{SC}$	0.05
	IIF^+	0.05
	$\mathsf{NN}^+\mathsf{IIF}^+$	0.04
	$\mathsf{IIF}^+\mathsf{NN}^+$	0.04
	$\mathsf{SCF}^+\mathsf{NN}^+$	0.03

The $\mathsf{SCF}^+\mathsf{NN}^+\mathsf{SCF}^+$ pattern found in the E_1I_0 and E_1I_1 categories, arises in class identifier names like `BuddyPluginPasswordException` found in Vuze which has the super class `BuddyPluginException`, where one or more nouns are inserted into the superclass name to indicate the specialisation.

Where both dimensions of inheritance are used, the E_1I_1 and E_1I_n categories, the most common name inheritance components are $\mathsf{NN}^+\mathsf{SCF}^+$, $\mathsf{II}\,\mathsf{SCF}^+$ and $\mathsf{NN}^+\mathsf{SC}$. The E_1I_1 category contains identifier names composed exclusively of words repeated from both the super class and implemented interface identifier names. An example of the $\mathsf{II}\,\mathsf{SCF}^+$ pattern is the JabRef class `FieldTextArea` which is a sub class of `JTextArea` and implements the `FieldEditor` interface.

The repetition of entire interface names as suffixes, the $\mathsf{NN}^+\mathsf{II}$ pattern, is absent from

the E_1I_n category despite being the most common pattern in both the E_0I_1 and E_0I_n categories. An example of the class name pattern is the `HeapRowLocation` class in Derby which implements the interface `RowLocation`. Many of the instances of the IINN^+ pattern are examples of class identifier names ending in `Impl`, an abbreviation for implementation, such as `GroovyFeatureImpl` found in NetBeans. Also relatively common are class identifier names composed exclusively of fragments of an interface name, i.e. the pattern IIF^+. This can result from the convention followed by some developers and projects of using the letter `I` as a prefix to indicate an interface identifier name, e.g. `IDialogSettings` found in Eclipse, where the implementing class is named `DialogSettings` (The Eclipse Foundation, 2007). Other examples of the IIF^+ pattern include the class `JNDIResource`, found in JBoss, that implements the interface `JNDIResourceMBean`, and the Java library class `RMISocketFactory`, which implements the interfaces `RMIClientSocketFactory` and `RMIServerSocketFactory`.

RQ 5 asked how developers incorporate super class and interface names in class names. The answer is complex. The majority of class names that extend another class or implement an interface incorporate some element of one or more of the name of the super class or super type. In general, the preference is that the class adds one or more nouns as a prefix to the name or name component repeated from the super class or super type. Where the class both extends another class and implements an interface, elements of the super class name tend to be more commonly incorporated in the class name, than the interface name. However, there appear to be no obvious rules for the observed patterns that can be derived from studying a single generation of inheritance.

The investigation of class name structure undertaken to answer RQ 4 found that 12% of class names are not nouns or noun phrases. 2% of class names were found to be verb phrases and the remaining 10% consist of variety of simple and complex phrasal structures that, like verb phrases, are not specified in naming conventions. In answering RQ 5 dominant patterns of reuse of the name or fragments of the names of immediate super classes or super types were identified. A range of less commonly used patterns of name reuse were also found. So far I have not tried to understand why developers might use the less common phrase structures and patterns of name reuse in class names, and whether there is any significance to the less common and 'unconventional' patterns of naming. I explore this question in a case study of the class names found in FreeMind.

5.3.3 FreeMind

FreeMind was selected as the subject for the case study for two reasons. Firstly, FreeMind is a mature Java GUI application of modest size (652 classes and 41 KSLOC) that implements a wide range of functions including editing and export to a variety of document formats. And, secondly, because I am a long-term user of FreeMind and am familiar with its functionality.

The frequency of the most common grammatical patterns in the 652 class identifier names found in FreeMind can be seen in Table 5.6. The four most common patterns are the same as those found with the greatest frequency for all projects (Table 5.1), and were found with similar frequencies apart from the VBNN^+ pattern where the frequency for FreeMind was almost double that for all the projects analysed. In total the four most common grammatical patterns describe 91% of the class identifier names in FreeMind.

Table 5.6: Common part of speech patterns and frequencies for FreeMind

Pattern	Absolute Frequency	Relative Frequency
NN^+	459	0.70
$\mathsf{JJ}^+\mathsf{NN}^+$	81	0.12
VBNN^+	39	0.06
$\mathsf{NN}^+\mathsf{JJ}^+\mathsf{NN}^+$	20	0.03

The distribution of class identifier names in the inheritance categories for FreeMind given in Table 5.7 show a greater proportion of classes in FreeMind either extend a super class or implement one or more interfaces than is observed across all the projects analysed (see Table 5.2). Indeed, 66% of classes in FreeMind extend a super class, as opposed to 58% average across the corpus.

Table 5.7: Distribution of inheritance categories for FreeMind

Category	Absolute Frequency	Relative Frequency
E_0I_0	82	0.13
E_0I_1	128	0.20
E_0I_n	14	0.02
E_1I_0	318	0.49
E_1I_1	78	0.12
E_1I_n	32	0.05

Table 5.8 shows the distribution of common grammatical forms of class identifier names that include words derived from the super class or implemented interfaces for each of the inheritance categories in FreeMind. While many patterns are common to the same categories in Table 5.8 and Table 5.5, there are key differences that indicate the existence of project-specific naming conventions in FreeMind. For example, the two most frequently occurring patterns in the E_0I_1 and E_0I_n categories overall, $\mathsf{NN}^+\mathsf{II}^+$ and $\mathsf{NN}^+\mathsf{IIF}^+$, are much more common in FreeMind. Furthermore, the pattern $\mathsf{NN}^+\mathsf{IIF}^+$, where a fragment of the interface name is repeated in the class identifier name, is the more frequent of the two in FreeMind and accounts for 29% of the class identifier names in the E_0I_n category.

Table 5.8: Common grammatical forms of class name component inheritance for FreeMind

Category	Grammatical Form	Relative Frequency
E_0I_1	$\mathsf{NN}^+\mathsf{IIF}^+$	0.27
	$\mathsf{NN}^+\mathsf{II}$	0.20
	$\mathsf{IIF}^+\mathsf{NN}^+$	0.05
	$\mathsf{JJ}^+\mathsf{NN}^+\mathsf{IIF}^+$	0.05
	IINN^+	0.04
E_0I_n	$\mathsf{NN}^+\mathsf{IIF}^+$	0.29
	$\mathsf{NN}^+\mathsf{II}$	0.21
	IIF^+	0.07
E_1I_0	$\mathsf{NN}^+\mathsf{SCF}^+$	0.35
	$\mathsf{NN}^+\mathsf{SCF}^+\mathsf{NN}^+$	0.07
	$\mathsf{NN}^+\mathsf{SC}$	0.06
	$\mathsf{SCF}^+\mathsf{NN}^+$	0.05
	$\mathsf{JJ}^+\mathsf{NN}^+\mathsf{SCF}^+$	0.04
	$\mathsf{VB}\,\mathsf{NN}^+\mathsf{SCF}^+$	0.04
	$\mathsf{JJ}\,\mathsf{SCF}^+$	0.04
E_1I_1	$\mathsf{NN}^+\mathsf{SCF}^+$	0.38
	$\mathsf{IIF}^+\mathsf{SCF}^+$	0.09
	IISCF^+	0.08
	$\mathsf{VB}\,\mathsf{NN}^+\mathsf{SCF}^+$	0.05
	$\mathsf{SCF}^+\mathsf{NN}^+$	0.05
E_1I_n	$\mathsf{NN}^+\mathsf{SCF}^+$	0.28
	$\mathsf{SCF}^+\mathsf{NN}^+$	0.09

The $\mathsf{NN}^+\mathsf{SCF}^+\mathsf{NN}^+$ pattern observed with a frequency of 7% in the E_1I_0 inheritance category in FreeMind occurs with much lower frequency across the projects analysed. Inspection of the instances of the $\mathsf{NN}^+\mathsf{SCF}^+\mathsf{NN}^+$ pattern reveals a local naming convention, an example

of which is the class `MindMapCloudModel` that extends the class `CloudAdapter`. The pattern occurs mainly in a small cluster of packages where it is used for classes that form fundamental components of FreeMind's mind maps. In this particular case, the `CloudAdapter` class is the sole implementer of the `MindMapCloud` interface, which suggests `CloudAdapter` could, perhaps, be renamed `MindMapCloudAdapter` to illustrate the lineage more clearly.

Another feature of FreeMind is the relative prominence of class names with grammatical patterns beginning with a verb in the E_1I_0 and E_1I_1 categories. Identifier names of the $\mathsf{VB\,NN^+SCF^+}$ pattern are mainly of the form $\mathsf{VB\,NN^+}$, when the origin of component words is ignored, and, as already noted, FreeMind has a relatively high frequency of identifier names of this general pattern (See Table 5.6). As discussed below, many of the classes responsible for handling user initiated actions in the GUI start with a verb. The $\mathsf{VB\,NN^+SCF^+}$ pattern is also prominent in the E_1I_1 category, where it is notable that the $\mathsf{NN^+SCF^+}$ pattern occurs with a much higher frequency in FreeMind (38%) than amongst all the projects analysed (15%). The $\mathsf{IIF^+}$ pattern found in the E_1I_1 and E_1I_n categories for all the projects surveyed are not found in the same categories for FreeMind. Where components from implemented interface identifier names are incorporated into class names in the E_1I_1 and E_1I_n categories in FreeMind so are fragments of the super class identifier name.

In addition to the four most common grammatical patterns given in Table 5.6, further patterns were identified accounting for a total of 53 class names (see Table 5.9), or a relative frequency of 0.08. The most common grammatical pattern amongst this group was $\mathsf{VB\,NN^+IN\,NN^+}$, where IN represents a preposition. Each of the 53 unconventionally named classes was inspected to understand whether the name used was appropriate and a clear reflection of the role the class plays in the application. If the name did not meet those criteria, I explored possible name refactorings that adhered to the established naming conventions found in the project and were permitted within the same namespace. I also considered whether the class might be refactored into classes that could be named conventionally.

The majority of the 53 classes inspected represented actions taken by the user in the GUI, or coordinating events such as automatically saving all open files. On the whole these class identifier names clearly described the role of each class and were not prone to name refactoring.

Some class names were identified that could be refactored to more conventional identifier

Table 5.9: Classes inspected in FreeMind

Package	Class name	Refactor Name	Refactor Class	Comment
accessories.plugins	ExportToImage	No	No	Describes action initiated in UI
	ExportToOoWriter	No	No	Describes action initiated in UI
	ExportWithXSLT	No	No	Describes action initiated in UI
	FitToPage	No	No	Describes action initiated in UI
	JumpToMapAction	No	No	Describes action initiated in UI
	MyFreemindPropertyListener	Yes	No	Rename AutomaticLayoutPropertyListener
	SaveAll	No	No	Describes action initiated in UI
	UnfoldAll	No	No	Describes action initiated in UI
accessories.plugins.dialogs	ArrayListTransferable	Yes	No	Rename TransferableArrayList
accessories.plugins.time	ReplaceAllInfo	No	No	Local naming convention
	ReplaceSelectedInfo	No	No	Local naming convention
accessories.plugins.util.xslt	FileChooseListener	Yes	No	Rename FileChooserListener
freemind.common	ThreeCheckBoxProperty	Yes	No	Rename TristateButton
freemind.controller	AboutAction	No	No	Describes action initiated in UI
	DisposeOnClose	No	No	Wraps Swing GUI action
	HideAllAttributesAction	No	No	Describes action initiated in UI
	MoveToRootAction	No	No	Describes action initiated in UI
	ShowAllAttributesAction	No	No	Describes action initiated in UI
	ShowSelectionAsRectangle	No	No	Describes action initiated in UI
	ZoomInAction	No	No	Describes action initiated in UI
	ZoomOutAction	No	No	Describes action initiated in UI
freemind.controller.filter	CreateNotSatisfiedConditionAction	No	No	Local naming convention
freemind.controller.filter.condition	AttributeCompareCondition	No	No	Local naming convention
	AttributeExistsCondition	No	No	Local naming convention
	AttributeNotExistsCondition	No	No	Local naming convention
	ConditionNotSatisfiedDecorator	No	No	Local naming convention
	IgnoreCaseNodeContainsCondition	No	No	Local naming convention
	NodeCompareCondition	No	No	Local naming convention
	NodeContainsCondition	No	No	Local naming convention
	NoFilteringCondition	No	No	Local naming convention
freemind.extensions	AllDestinationNodesGetter	No	No	Describes action initiated in UI
freemind.main	StdOutErrLevel	No	Yes	Remove member class
freemind.modes	NodeDownAction	No	No	Describes action initiated in UI
	SaveAsAction	No	No	Describes action initiated in UI
freemind.nodes.common	CommonToggleFoldedAction	No	No	Describes action initiated in UI
freemind.modes.mindmapmode	DoAutomaticSave	No	No	Time-based backup task
	ExportBranchToHTMLAction	No	No	Describes action initiated in UI
	ExportToHTMLAction	No	No	Describes action initiated in UI
	SetImageByFileChooserAction	No	No	Describes action initiated in UI
	SetLinkByFileChooserAction	No	No	Describes action initiated in UI
freemind.modes.mindmapmode.actions	AddLocalLinkAction	No	No	Describes action initiated in UI
	ChangeArrowsInArrowLinkAction	No	No	Describes action initiated in UI
	NodeUpAction	No	No	Describes action initiated in UI
	RemoveAllIconsAction	No	No	Describes action initiated in UI
	SelectAllAction	No	No	Describes action initiated in UI
	SetLinkByTextFieldAction	No	No	Describes action initiated in UI
	UsePlainTextAction	No	No	Describes action initiated in UI
freemind.modes.mindmapmode.attributeactors	MyRenderer	Yes	No	Rename AttributeTreeCellRenderer
	ToggleAllAction	No	No	Describes action initiated in UI
freemind.modes.mindmapmode.dialogs	EdgeWidthBackTransformer	Yes	No	Rename StringToEdgeWidthTransformer
freemind.modes.viewmodes	CommonToggleChildrenFoldedAction	No	No	Describes action initiated in UI
freemind.view.mindmapview	Selected	Yes	No	Rename SelectedNodes
freemind.view.mindmapview.attributeview	MyFocusListener	Yes	No	Rename AttributeTableFocusListener

names. Three were member classes with identifier names prefixed with `My`, which gives the reader little information about the origins of the class, or the detail of the functionality they might expect to encounter. For example the class `MyRenderer`, a member class of the class `ImportAttributesDialog` in the package `freemind.modes.mindmapmode.attributeactors`, is responsible for rendering the cells of a tree used for display in the dialogue. Other member classes of the same class have clear identifier names reflecting the detail of their purpose, e.g. `AttributeTreeNodeInfo`. To make the class name consistent with the other member classes, and to improve clarity, the name might be refactored to `AttributeTreeCellRenderer`, which

adheres to the common NN^+ pattern.

The class `ArrayListTransferable` is designed to protect an array list from modification while it is transferred between two objects. The adjective has been placed after the noun phrase it is intended to modify, and the name could be refactored to `TransferableArrayList`, which is both clearer and conforms to the common $\mathsf{JJ}^+\mathsf{NN}^+$ pattern.

The class `ThreeCheckBoxProperty` is a GUI component that implements a button used in dialogues where the user has a number of settings available. The button cycles through three states when clicked, which are represented by a plus sign, a minus sign and an empty box and have the meanings change property, remove property and ignore, respectively. The GUI component does not look like a check box or behave like one, so that aspect of the name appears to be incorrect. It is used in properties dialogues, but *property* in isolation does not adequately represent its usage context. The common name for this type of widget is a *tri-state checkbox*, so the most appropriate name refactorings are `TristateCheckBox` or `TristateButton`, both of which conform to the $\mathsf{JJ}^+\mathsf{NN}^+$ pattern, and do not contain extraneous detail about the type of dialogue in which the component is used.

I also found one instance of a spelling mistake: a class named `FileChooseListener`, which should have been `FileChooserListener` to be consistent with the name of the Swing `JFileChooser` instance it creates. The spelling mistake was identified as the result of the PoS tagger recording `Choose` as a verb, thus giving the identifier name the pattern $\mathsf{NN}^+\mathsf{VB}\,\mathsf{NN}^+$, which is relatively uncommon. The remaining four instances of this pattern in FreeMind are a part of a local naming convention in the `freemind.controller.filter.condition` package. Most classes extend `NodeCondition` or one of its subclasses and follow a consistent naming scheme based on their position in the hierarchy. A feature of the naming scheme is the insertion of a verb between the two component words of the super class. The classes support a filtering mechanism that selects particular nodes for display, e.g. the class `NodeContainsCondition` is used to test whether a node contains a particular condition or attribute. I don't consider the class identifier names to be a problem for program comprehension because the unconventional naming is used consistently, and the classes, on inspection, appear to function as described. However, the awkward nature of the naming pattern may be a design smell (a poor solution to a common design problem that compromises source code quality and maintainability (Moha *et al.*, 2010; Suryanarayana *et al.*, 2014)) and that using

a more conventional design, e.g. the visitor pattern (Gamma *et al.*, 1995), should result in more conventional class identifier names.

The class `StdOutErrLevel` is a candidate for refactoring. `StdOutErrLevel` is a member class of `freemind.main.StdFormatter` used in logging and was identified because the intended abbreviations of 'output' as 'out' and 'error' as 'err' are also words. On initial reading of the name, it appears to be an abbreviation of *standard output error level*, or *stdout error level*. On inspection, the `StdFormatter` class is responsible for formatting log records for FreeMind, and `StdOutErrLevel` is used to assign the logging threshold for the *stdout* and *stderr* streams. The developers combine the task of setting the logging thresholds for two streams into the same object, before specifying each output stream in the call to the constructor (see Figure 5.2). There are two solutions: either the class name is refactored to remove the reference to both standard streams, or the class, which wraps the `java.util.logging.Level` class without adding any functionality, is removed and replaced by direct calls to the `Level` library class. The latter is the preferable solution as it results in a less cluttered class that is easier to read.

```
/**
 * Level for STDOUT activity.
 */
final static Level STDOUT =
    new StdOutErrLevel("STDOUT", Level.WARNING.intValue()+53);

 /**
  * Level for STDERR activity
  */
 final static Level STDERR =
    new StdOutErrLevel("STDERR", Level.SEVERE.intValue()+53);
```

Figure 5.2: Partial listing from freemind.main.StdFormatter

`EdgeWidthBackTransformer` is one of a group of member classes of `StylePatternFrame` in the package `freemind.modes.mindmapmode.dialogs` that transform strings to widths and back again. The class performs the inverse function of the class `EdgeWidthTransformer` and invokes the method `transformStringToWidth`. A clearer and more consistent class name might be `StringToEdgeWidthTransformer`. However, the identifier name remains

unconventional.

5.3.4 Threats to Validity

As with any empirical study there are threats to validity. In this case threats to validity concern construct and external validity. Internal validity is not considered because I make no claims of causality. Also, I have not used any statistical inference, so do not consider statistical conclusion validity.

Construct Validity The key threat to construct validity is the accuracy of the PoS tagger used in the experiment. The Stanford PoS tagger's test mode for the tagger model I trained reports an accuracy of 95% for individual words and 85% for whole identifier names. The sources of error include words that commonly have more than one part of speech, abbreviations and unknown words. However, despite the error rate I successfully identified unconventionally constructed identifier names that were candidates for refactoring as well as a possible design smell. An associated threat is that I hand tagged the training and testing corpora, which is a possible source of bias.

The consequence of collapsing the Penn Treebank PoS tags is that 50 proper nouns and 15,049 plural nouns are included in the NN tag, and a total of 1,393 verb forms are included in the VB tag. These account for 13% of class identifier names containing nouns and 24% of those containing verbs. Only 0.2% of adjectives are tagged as JJR and JJS. While this approach hides some detail, it allows the observation of general forms of class identifier names.

External Validity The corpus of class names consists of the class names recorded in INVocD (Chapter 3). The 60 projects in INVocD are drawn from a variety of domains to reduce the influence of domain specific identifier names or naming styles. There is considerable variation in the proportions of class and identifier based inheritance used in the projects. Accordingly, while the observations of the patterns of the reuse of component words from the names of super classes and implemented interfaces are reliable for the corpus analysed, caution should be exercised when extrapolating the proportions of these patterns to other projects.

5.4 Discussion

I employed two methods of analysing of class identifier name structure. The first relies solely on the parts-of-speech used in the class names, and the second considered the origins of the component words in the names of the super class and implemented interfaces. I found that more than 90% of class identifier names can be described using four simple grammatical patterns. Despite the advice given in naming conventions (Gosling *et al.*, 2014; Vermeulen *et al.*, 2000) that class identifier names should be nouns or noun phrases, I found that a proportion — around 2% — incorporate verbs and describe actions, rather than entities. Inspection of a sample of the class identifier names containing verbs found that many were related to actions initiated by the user in GUI environments.

Naming conventions offer no advice on whether, or how to incorporate information from super class or interface names in class identifier names. Praxis, in the Java library, for example, is for some class and interface hierarchies to retain part of the class identifier name through the inheritance hierarchy. My analysis of the origins of component words in class identifier names found that 70–80% of classes that extend a superclass or implement an interface include one or more words repeated from the super class or implemented interface name (See Tables 5.4, 5.5 & 5.8).

In general, class identifier names repeat fragments of the super class or interface name, rather than the entire name, and it is words found in the super class names that are repeated with the higher frequency. Manual inspection showed that fragments are mostly derived from the latter, or right hand, part of the super class or interface name. Though a common pattern found in exception classes involves the insertion of words in the super class name. I was unable to identify any obvious mechanisms from single generation inheritance that explain how a decision is made to repeat either part or all of the super class or interface name. However, I anticipate that it will be possible to derive heuristics from inheritance trees where name fragments are repeated over more than one generation.

The repetition of components of super class and implemented interface identifier names in some class identifier names appears to violate the conciseness rule of Deißenböck and Pizka's system of concise and consistent naming (Deißenböck and Pizka, 2006). A *concise* name is one that unambiguously represents a given concept within a program. For example, the identifier names `position` and `absolutePosition` would break the conciseness rule because

the concept of 'position' *contains* the concept 'absolute position'.

Lawrie *et al.* (2007a) used a syntactic methodology to investigate violations of concise and consistent naming. They defined *Type I* and *Type II* syntactic violations, both of which imply that the conciseness rule has been broken, and may indicate a failure of the consistency rule. A Type I violation occurs when an identifier name is repeated entirely within another identifier name; and a Type II violation occurs when an identifier name is similarly contained by two or more others. Type I violations occur in a single generation of inheritance where a class identifier name includes the *whole* of the name of either the super class or an implemented interface.

In FreeMind, for example, 89 (14%) of the class identifier names surveyed are Type I violations. Some class names appear to be genuine violations of the conciseness rule, but most are part of the process of creating program concepts through inheritance. The same is true of the Type II violations I identified. In the latter case the false positives are typified by classes that implement a common interface or base class, e.g. `SortedComboBoxModel` and `ClonedComboBoxModel` both implement the Swing interface `ComboBoxModel`, and describe the concept hierarchy clearly. Lawrie *et al.* (2007a) suggested that parts-of-speech may help discriminate between identifier names that represent new concepts and those that are genuine violations. The adjectives in the example fulfil that role, but further study is required to confirm the viability of the method.

Importantly, around 20–30% of class identifier names in each inheritance category were found not to incorporate any words derived from the identifier names of the extended super class or implemented interfaces. Further investigation is needed to identify the occasions on which names are incorporated and those when they are not. One approach is to derive rules of name inheritance from existing behaviour within a code base, which may have a practical application by alerting a developer to an unusual repetition of a component word in a class identifier name, or the omission of a component that is commonly repeated. For example, the Java library interface names `Cloneable` and `Serializable` are rarely incorporated in the identifier names of implementing classes, because they describe functionality to be incorporated into a class rather than a type definition. While such a solution is attractive, it does not explain when and why super class and interface name components are repeated. A more detailed approach, analysing identifier names in terms of their grammatical structure,

their semantics, and their role or position in the inheritance hierarchy, may result in methods that predict the circumstances under which component words are repeated and which words should be repeated.

The common class identifier naming patterns are familiar mechanisms developers use to communicate ideas. Høst and Østvold demonstrated that the link between Java method identifier names is sufficiently strong that poor quality or misleading names can be identified and candidate refactorings suggested (Høst and Østvold, 2009). By identifying candidates for renaming, such as `ThreeCheckBoxProperty`, and a possible design smell, I show that practical results may be achieved through the recognition of unconventionally constructed Java class names. However, while identifier names may reflect the implementation of a class, conventionally structured class identifier names are not a guarantee of flawless design.

Knowledge of the common conventional grammatical patterns of class identifier naming can be incorporated in IDE-based tools to support the creation of class identifier names that are more readily understood by developers, and that conform to project standards determined by software project managers. Such a tool could alert the developer to an unconventionally structured name and, eventually, recommend possible improvements, including the incorporation of words used in the super class and interfaces, if any. Developers new to a project would have a ready made style guide to support the creation of class identifier names that are familiar to existing colleagues. The same knowledge can be leveraged to provide quality assurance for software project managers that is an improvement on the functionality of current tools. For example, CheckStyle[5] provides only very basic checks of the typographical structure of identifier name, e.g. whether a class name begins with an upper case letter.

For software maintainers new to a project, a tool that extracts the project's class naming conventions provides an overview of how the project's developers encode information in identifier names, particularly the extent to which names reflect inheritance help them understand the code base. Such information supports program comprehension by identifying which component words are repeated in inheritance trees and can be used to identify related classes and help target lexical searches.

[5] `http://checkstyle.sourceforge.net/`

5.4.1 Future Work

In this chapter, I have examined how whole and part class and interface names are incorporated into class names for a single generation of inheritance. While the study of inheritance within a single generation demonstrates a great deal about the way in which developers incorporate elements of names in the names of subclasses, it raises questions about the reasons developers reuse elements of names, and possible relationships to the patterns of inheritance used. Further investigation of the reuse of name components in inheritance trees could lead to deeper understanding of why particular name elements are reused, and under what circumstances. To support that initiative, detailed investigation of inheritance trees is also needed to understand the range of use of inheritance by developers in terms of depth (generations) and structure and complexity, and the relationship between the use of inheritance and patterns of class naming. Furthermore, any investigation would need to be sensitive to changes to the Java language, especially the introduction of methods to interface definitions in Java v8, which allows the use of mixin classes in Java for the first time.

Suggestions of avenues of research that may improve the PoS tagging methodology are discussed in the following chapter (Section 6.4.1).

5.5 Summary

Through the analysis of class identifier names extracted from the 60 Java open source projects in `INVocD` and a case study of FreeMind, I have taken a step towards a deeper understanding of Java class identifier naming conventions used in practice. In this chapter I make three contributions to the understanding of class names:

1. I identify the common grammatical structures of Java class identifier names found in praxis, and their distributions.

2. I identify the patterns by which component words from the super class or implemented interfaces are repeated in class identifier names, and record their distributions.

3. I show, for the example of FreeMind, how the detailed knowledge of project-specific uncommon class identifier naming patterns can be put to practical use to help detect poor class names and design smells and thereby improve program comprehension and

design.

The study makes a fourth, technical, contribution by demonstrating that it is possible to train a model for an existing PoS tagger, designed to process sentences, that can be used to tag identifier names.

My analysis can be applied in practical software engineering tools to support identifier naming by making developers aware of the naming conventions used in the project they are working on. A tool could offer guidance that supports the creation of more commonly recognisable names, such as indicating when an unconventional form is being used, or making recommendations of possible identifier names during development. The same knowledge can also be applied by software project managers to configure the developers' tools with project-specific standards, and in tools for identifier name quality assurance that are a considerable improvement on current tools that check the typographical form of names.

In the following chapter I investigate the structure of Java reference names and extend the analytical techniques employed in the investigation of class names. Reference names constitute the majority of name declarations in source code and contain a much greater variety of content types than class names. As well as applying the PoS tagging method developed for class names to those reference names that contain natural language content, I use a parser to identify the phrasal structures found in reference names.

Chapter 6

Phrasal Analysis of Reference Identifier Names

Following the investigation of class names described in the previous chapter, attention in this chapter turns to reference (field, parameter and variable) names, the second large group of names identified for investigation in the principal research question. Reference names constitute 52% of the unique names, and 69% of all declarations found in the 60 projects recorded in `INVocD` and are a potentially rich source of information for the tools that support program comprehension, including code search and feature location. The investigation of reference names applies the technique of training a PoS tagger model on identifier names, developed to analyse class names, to the analysis of reference names.

Identifier naming conventions (Gosling *et al.*, 2014; Vermeulen *et al.*, 2000) provide developers with guidelines for composing names. The guidelines can be complex, particularly for reference names, providing developers with a wide choice of the form of name they create, including: phrase-like names containing words, abbreviations and acronyms; isolated abbreviations; acronyms derived from type names; and specialised, single-letter abbreviations for generic or short-lived identifiers with well-understood roles. Conventions, however, are not hard-and-fast rules and developers can create identifier names as they please. Consequently the readers of source code, including program comprehension tools that rely on the content of names, have only an outline of the forms names might take, and the possibility of being surprised is great.

Liblit *et al.* (2006) identified, through observation, patterns of naming they referred to

as *metaphors* where they considered names to be phrasal utterances. Their metaphors are often used as a starting point by those analysing names for program comprehension (Hill, 2010; Abebe and Tonella, 2010; Gupta *et al.*, 2013). However, the extent to which Liblit *et al.*'s metaphors are used by developers has not been established for all species of identifier. Investigations of Java method names (Høst and Østvold, 2008) and class names (Chapter 5) show that the metaphors are used extensively and illustrate that developers also create names with unanticipated phrasal structures, both simple and complex (Section 5.4). Developers might be expected to be similarly inventive with reference names.

In this chapter I seek to establish the forms of reference names created by developers and the extent to which the various forms are used. As mentioned earlier, naming conventions suggest a variety of name content types, e.g. dictionary words, acronyms, and abbreviations. My first research question is:

> **RQ 6 What content types do developers use to create reference names, and to what extent is each content type used?**

As part of RQ 6, I want to know the extent to which recognised abbreviations are used, that might be readily comprehended by humans, but not by tools without abbreviation expansion.

The literature argues that the use of natural language phrases and metaphors can improve program comprehension. I wish to know:

> **RQ 7 What phrasal structures do developers use in reference names, and how are they related to Liblit *et al.*'s metaphors?**

6.1 Related Work

There are two principal themes of related research. The most closely related strand of research concerns the investigation and cataloguing of identifier name structure. The second strand concerns the 'pragmatic' grammars used in approaches developed to extract semantic information from identifier names. The latter strand was discussed in the preceding chapter (Section 5.1), and is only reviewed briefly in this section.

Influencing all but the earliest work on the natural language structure of names is Liblit *et al.*'s wide ranging treatise on identifier naming, which observed that names are 'pseudo-grammatical utterances' or phrase fragments (Liblit *et al.*, 2006). On the basis of program-

ming experience and observation, but without systematic quantification of their use in practice, Liblit *et al.* described *metaphors* for identifier names that reflect the role of the name. One metaphor is *data are things*, so that identifiers of data objects are named with nouns or noun phrases, and methods that behave as mathematical functions — typically a method that has no argument and returns a non-boolean value or object reference — are named with noun phrases that reflect the returned value (Liblit *et al.*, 2006), e.g. the method `size()` in Java collection classes.

The first grammar of identifier name structure was discovered by Caprile and Tonella (1999), who analysed C function names. The grammar described the majority of function names. The grammar was subsequently applied to refactor function names to make them more meaningful (Caprile and Tonella, 2000). A similar analysis of Java method names was undertaken by Høst and Østvold (2008) using a specially developed PoS tagger that also considered type names to be a separate part of speech, i.e. the PoS tagger also did some semantic processing prior to tagging. Høst and Østvold found a complex grammar with many 'degenerate' forms. They also found a relationship between the structure of a method name and the functionality of the method, sufficient to automate the detection of names that did not accurately describe the implemented methods (Høst and Østvold, 2009).

A survey of field names by Binkley *et al.* (2011) employed the default Stanford tagger model trained on the Wall Street Journal corpus to analyse C++ and Java field names containing only words, abbreviations and acronyms found in the SCOWL lists up to size 50. The survey used four templates to provide additional context for the PoS tagger, e.g. the tokens were followed by 'is a thing' to nudge the tagger towards treating the name as a noun. Three other templates were used that treated the name as a sentence, a list item, and a verb. The main aim of the survey lay in evaluating the efficacy of the method rather than undertaking an exhaustive survey of field name structure. However, it was found that 88% of names were PoS tagged correctly using the four templates.

A survey of class, method and field names in C++ and Java by Gupta *et al.* (2013) employs the technique of using WordNet (Fellbaum, 1998) to identify candidate PoS tags that were then used to determine the phrasal structure of names, instead of employing conventional PoS tagging techniques. A simplified set of PoS tags, closely following those devised by Hill (2010), are deemed sufficient for identifiers rather than the Penn Treebank. Gupta *et*

al. found that non-boolean field names are typically noun phrases, while boolean fields are generally verb phrases that ask a question. However, their study concerned the evaluation of their PoS tagger and does not quantify their finding on field name structure.

Lawrie *et al.* (2007b) undertook a statistical investigation of the differences in identifier name quality between 78 proprietary and open source projects written in different programming languages, and projects developed at different times over a 30 year period. Their measures of name quality include the relative proportions of dictionary words, abbreviations, and single-letter abbreviations in names, and the length of names as the number of tokens. The work reported in the remainder of this chapter differs in its intent, and the scope and nature of the analysis. I document the variation in composition of reference names only, rather than all names, and use finer-grained definitions of tokens with a focus on the level of processing the token might require. I give a per-species analysis of name composition to inform the suitable strategies to be adopted by the developers of program comprehension tools.

The literature describes a range of approaches to the extraction of semantic information from names. All rely on PoS tagging to help identify the structure of names. In some cases assumptions are made about the structure of names to simplify processing. The most comprehensive approach developed so far is adopted in the software word usage model (SWUM) developed by Hill (2010). SWUM uses a general grammar for all species of name to support semantic parsing. The grammar relies on a smaller set of PoS tags than the Penn Treebank and includes productions for noun, preposition and verb phrases. Hill does not quantify name structure, nor the coverage of the grammar for the various species of name.

Abebe and Tonella (2010) developed the system of using templates (a technique adopted by Binkley *et al.* (2011)) to try to create a statement or sentence that provides additional context to support a PoS tagger. The approach uses `Minipar` to parse the resulting name and template combination, which is rejected if `Minipar` cannot identify an element in the sentence. The approach uses a limited number of templates — e.g. 5 for field names — that can only be used to identify a few types of single phrase names. The technique is intended to support concept identification, and was subsequently used to extract ontologies from source code (Abebe and Tonella, 2011).

6.2 Methodology

The strong typing of Java offers two features that simplify the identification of the role of a reference name. Firstly, in Java all boolean identifiers are declared using the `boolean` primitive or the `Boolean` object type, unlike C for example, where numeric values may be used as booleans. This distinguishes all boolean identifiers allowing investigation of Liblit *et al.*'s observation that they differ in structure to non-boolean names (Liblit *et al.*, 2006). Secondly, analysing the types of reference names in Java is feasible because, with the exception of the reflection API, there are no function pointers in Java, and there is a clear distinction between actions and entities. All source code investigated pre-dates the introduction of method references in Java 8.

6.2.1 The Dataset

The corpus investigated is the bag, i.e. a set with duplicate elements, of all reference name declarations in `INVocD`: 626,262 field, 1,556,931 formal argument, and 1,319,071 local variable declarations. Declarations are examined to distinguish instances of the same name declared with different species or types.

Each declaration indicates the need for a name, and the developer is free to choose any name they desire. If the developer reuses a particular name, it indicates preference for certain identifier forms, phrasal structures or metaphors. The reuse of names in the corpus is substantial. The 3,502,264 declarations consist of only 272,228 unique field names, 81,201 unique formal argument names and 169,428 unique local variable names. Reuse rates in the corpus are thus 2.3, 19.2 and 7.8 times for field, formal argument and local variable names respectively.

To capture such preferences, the corpus is a bag instead of a set, i.e. *all* declarations are considered, even of the same name with the same type and species. In this way, for a program that declares 100 integer local variables, one named `xpto` and the rest `i`, 99% of the declarations are expected name forms (namely `int i`), whereas considering only unique names or unique declarations would lead to a distorted figure of 50%, when in fact the developers made 100 choices, only one of which deviated from established guidelines.

For names to be considered phrases they should, in general, consist of sufficient 'words' to form a phrase and not so many as to form multiple phrases. Lawrie *et al.* (2007b) found

that Java names have 3.4 tokens on average. These are mean values for all the unique names found in the code they investigated: there was no breakdown by name species and no measure of central tendency. `INVocD` provides each name's tokens, as identified by `INTT` (Chapter 4). Table 6.1 shows the distribution of the length of the unique reference names in `INVocD` as the number of tokens. Field name length is similar to Lawrie *et al.*'s mean, while formal argument and local variable names tend to be shorter. The longest name, consisting of 39 tokens, is one of a number of long names given to some strings in Eclipse used as keys in resource bundles. The name is a field of type `java.lang.String`:

```
ThreadReferenceImpl_
Unable_to_pop_the_requested_stack_frame_from_the_call_stack__
Reasons_include__
The_requested_frame_was_the_last_frame_on_the_call_stack__
The_requested_frame_was_the_last_frame_above_a_native_frame__
12
```

The name consists of a variety of components including a class name, a number and sentences. Custom typographical conventions are applied with the use of single underscores to separate words and double underscores to separate most of the major components. Names like this example are composed of multiple phrases and are discussed in Section 6.4.

Table 6.1: Distribution of length (in tokens) of unique reference names

	Field	**Formal Argument**	**Local Variable**
Minimum	1	1	1
1st Quartile	2	2	2
Median	3	2	2
Mean	3.1	2.3	2.4
3rd Quartile	4	3	2
Maximum	39	10	12

As mentioned earlier, boolean names are examined separately. The proportion of unique reference names declared using the Java types `boolean` or `Boolean` in the subject projects is shown in Table 6.2. There is considerable variation between the projects. I found that between 1.1% and 15.5% of unique field names are declared as booleans in the projects investigated, and over 20% of unique formal arguments in ANT, OpenProj and Vuze.

Table 6.2: Distribution of proportions of unique boolean reference names

	Field	Formal Argument	Local Variable
Minimum	.011	.024	.017
1st Quartile	.067	.079	.046
Median	.083	.113	.063
Mean	.086	.112	.063
3rd Quartile	.102	.136	.079
Maximum	.155	.218	.118

Java does not allow the use of punctuation in names, such as the use of apostrophes to identify possessive forms and contractions of negated modal verbs, like "can't". Consequently, where possessive forms and negated modal verbs are used in names, e.g. `ER_CANT_CREATE_URL` (Xalan), an apostrophe-less form is used. Whilst negated modal verbs are not extensively used, they are easily recognised and expanded, allowing them to be tagged correctly (Section 6.2.3) to reduce noise. I therefore expanded all non-apostrophised negated modal verbs to their two-word form, prior to any further processing. For example, 'shouldnt' is expanded to 'should not' and 'wont' to 'will not'. Although 'cant' (meaning hypocritical and sanctimonious talk, among other things) and 'wont' (meaning accustomed) are English words, I still interpret them as negated modal verbs, as it is the most likely use in identifier names. No attempt was made to identify or expand possessive forms of nouns, a task which is left for future work because potential solutions rely on knowledge of the likely phrasal structure of a name to detect possible possessive nouns.

6.2.2 Partitioning Names

Research questions RQ 6 and RQ 7 concern the content and phrasal structure of names. To answer RQ 7 it is necessary to identify those names that contain tokens that are not susceptible to phrasal analysis. Naming conventions, such as those in JLS ('Java Language Specification' (Gosling *et al.*, 2014)) and EJS ('The Elements of Java Style' (Vermeulen *et al.*, 2000)), direct developers to use a mixture of well understood single letter names, acronyms derived from the type name, other acronyms[1], abbreviations, words, and multi-token names that combine the previous three categories. Tokens containing digits may be known acronyms, such as 'MP3', otherwise they are categorised as unrecognised. Accordingly, I partition reference

[1]The definition of *acronym* includes initialisms such as HTML.

name declarations into the following bags for each species:

- C contains *ciphers*, i.e. well-known or conventional single letter abbreviations (Table 6.3);
- T contains acronyms derived from type names;
- P contains names consisting only of 'processed' content types: dictionary words, known technical terms and acronyms, and an optional redundant prefix (discussed in Section 6.2.3);
- U contains names with at least one token that is an 'unprocessed' content type, i.e. an abbreviation or an unrecognised token.

Table 6.3: Ciphers and their corresponding types

Cipher(s)	**Type(s)**	**Source**
`b`	`byte`, `Byte`	JLS
`c`	`char`, `Character`	JLS, EJS
`d`,`e`	`char`, `Character`	EJS
`d`	`double`, `Double`	JLS
`e`	`Exception`	JLS
`f`	`float`, `Float`	JLS
`g`	`Graphics`	EJS
`i`,`j`,`k`	`int`, `Integer`	JLS
`l`	`long`, `Long`	JLS, EJS
`o`	`Object`	JLS, EJS
`s`	`String`	JLS, EJS
`v`	a value of some type	JLS
`x`,`y`,`z`	any numeric type	EJS

Names are partitioned in the following order:

1. If a name consists of a single token, the partitioning process starts with step 2, otherwise the process begins with step 4.
2. A name is first tested to determine whether it is a cipher from Table 6.3, and whether it is of the correct type. Table 6.3 shows that I have widened the JLS and EJS definitions of permitted types to include the Java v5 classes that wrap primitive types such as `Integer`, so that the declarations `for ( int i; ...)` and `for ( Integer i; ...)` are both considered ciphers.

3. If a name is not a cipher it is checked to determine if it is a type acronym as suggested in the JLS, i.e. an initialism derived from the declared type name, e.g. `FileWriter fw`. Applying these tests places declarations of `Iterator i` in T.

4. Names not meeting the requirements for C and T are partitioned into P and U: those that consist of recognised prefixes, words, acronyms and technical terms are assigned to P, the rest to U.

To support the creation of the P and U partitions I adapted an open source spellchecking library to use multiple dictionaries and to report in which dictionary a textual term was found, and whether those not found might be spelling errors[2]. The dictionaries are created using publicly available word lists. The SCOWL word lists (Atkinson, 2004) (up to size 80), that are used to create the dictionaries for GNU Aspell[3], were partitioned into separate dictionaries of words, abbreviations and acronyms. Additional dictionaries of abbreviations and acronyms were created from the lists used in the AMAP project[4] (Hill *et al.*, 2008), and technical terms, abbreviations and acronyms that were collected during the creation of `INTT` (Chapter 4).

Some projects use *redundant prefixes*: a single letter at the beginning of a reference identifier name that either indicates its role or type, e.g. some field names are prefixed with `f` for 'field' or `m` for 'member'. Single letters are also occasionally used to represent primitive types in a manner similar to Hungarian Notation (Heller and Simonyi, 1991), including the letters `b`, `c`, `d`, `f`, `i`, `l` and `o`, standing for *boolean*/*byte*, *char*, *double*, *float*, *int*, *long* and *object* respectively. The proportion of field names with redundant prefixes varies according to the naming style adopted by project developers. In many projects these prefixes are not used, but in a few projects the use of prefixes is conventional, particularly to indicate the role of the name, e.g. the use of `f` to prefix mutable fields in JUnit. Redundant prefixes are ignored when partitioning name declarations, so, for example, `pPropertyName` (FreeMind) is a member of P and `fBefores` (JUnit) is a member of U.

Table 6.4 shows the distribution of declarations in each partition in the projects analysed. Most declarations are in P, e.g. at least 47.8% of each project's field name declarations use

[2]The `MDSC` spellchecking library is available from `https://github.com/sjbutler/mdsc/` and contains the word lists used to partition names.

[3]`http://aspell.net/`

[4]`http://msuweb.montclair.edu/~hillem/AMAP.tar.gz`

only English words, recognised acronyms, known prefixes and technical terms.

Table 6.4: Distribution of proportions of declarations in each partition

		Field	Formal Argument	Local Variable
C	**Minimum**	.000	.001	.012
	1st Quartile	.000	.043	.056
	Median	.001	.066	.075
	Mean	.002	.065	.087
	3rd Quartile	.002	.086	.110
	Maximum	.015	.193	.240
T	**Minimum**	.000	.002	.006
	1st Quartile	.001	.016	.036
	Median	.004	.033	.054
	Mean	.007	.037	.060
	3rd Quartile	.009	.050	.079
	Maximum	.043	.129	.209
P	**Minimum**	.478	.267	.418
	1st Quartile	.748	.715	.642
	Median	.812	.781	.693
	Mean	.807	.767	.692
	3rd Quartile	.862	.083	.766
	Maximum	.961	.965	.876
U	**Minimum**	.039	.027	.045
	1st Quartile	.129	.081	.121
	Median	.172	.112	.155
	Mean	.185	.130	.161
	3rd Quartile	.238	.150	.183
	Maximum	.522	.711	.539

A consequence of the method used to create partitions is that all names containing spelling mistakes and readily understood neologisms — such as `fBefores` — not in the word lists will also be assigned to U. Therefore, U has names that may contain English words, but require further processing, such as abbreviation expansion, spellchecking and neologism checking. Such names would be a source of noise in the phrasal analysis.

I do not expand abbreviations. Abbreviations may have more than one plausible expansion and determining the correct one can require assumptions of the phrasal structure of names that pre-empts any investigation of that structure.

6.2.3 PoS Tagging

In the investigation of class names reported in the preceding chapter (Chapter 5) I analysed the composition of Java class names in terms of the parts of speech (PoS) of their component words using a model for the Stanford Log-linear PoS tagger (Toutanova *et al.*, 2003) I had trained on a corpus of Java class names. I used the model trained on class names with v3.4.1 of the Stanford tagger to PoS tag a set of Java field names. Manual inspection of the tagged field names showed an error rate around 28%, some 15% greater than observed when tagging class names. I decided to train a new model on field names to see if that performed better.

I extracted 30,000 unique field names at random from the database and manually tagged a training set of 29,894 field names. 106 names were discarded because they could not be PoS tagged. Typically the discarded names consisted of one or two abbreviations that were either ambiguous or unrecognised, or incomprehensible combinations of words and abbreviations or neologisms. Examples include `TRGDFTRT` (Derby), `icSigPs2CRD2Tag` (JDK), and `WEAVEMESSAGE_ANNOTATES` (AspectJ). A model for the Stanford PoS tagger was created using the training set. A further 5,000 field names were manually tagged to provide a test set. In addition a smaller test set of 1,000 boolean field names was created to measure the performance of the tagger model on boolean names.

Redundant prefixes were removed from names in the training and test sets because they have no parallel in natural language and thus cannot be reasonably tagged by the Stanford PoS tagger.

The test mode of the Stanford PoS tagger was used to evaluate the performance of the model when tagging the test set. The trained model tagged 85.4% of the field names in the test set correctly (94.5% of individual tokens) and 83.0% of the test set of boolean field names (93.2% of individual tokens).

Formal argument names appear similar in structure to field names. Indeed, in some naming styles, formal arguments for constructors and mutator methods have identical names to fields (Vermeulen *et al.*, 2000). Rather than undertake the potentially unnecessary work of creating a PoS tagger model for formal arguments, I manually tagged a test set of 5,000 formal argument names extracted at random from the database. The field name PoS tagger tags 91.7% of formal argument names in the test set correctly and 96.0% of individual tokens.

As with field names, any redundant prefixes were removed from formal arguments used

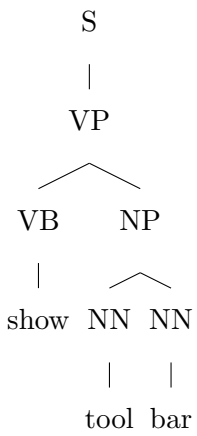
in the test set. The prefixes `p` and `m` are used in combination to distinguish between parameters and members with the same name, e.g. the constructor of `StandardPropertyHandler` (FreeMind) has the formal argument `pPropertyName` used to set the field `mPropertyName`.

Local variable names also appear to have a similar structure to field and formal argument names. The process was repeated to create a test set of 4,984 local variable names. According to the Stanford PoS tagger test mode, 90.3% of local variable names and 95.4% of individual tokens in the test set were tagged correctly.

After the tests, the PoS tagger model trained with field names was used to tag all names in the P bags of field, formal argument and local variable name declarations. Where a redundant prefix was found, it was removed and the remainder of the tokenised name tagged by the PoS tagger. The prefix was then added back to the beginning of the tagged string with the tag *RD* for ReDundant.

6.2.4 Phrasal Analysis

I used the Stanford Parser v3.4.1 (Klein and Manning, 2002) to identify the phrasal structure of names in the P partition to answer RQ 7. The Stanford Parser analyses a PoS tagged string and outputs a phrase structure tree. For example, the name `showToolBar` (BlueJ) is PoS tagged as show/VB tool/NN bar/NN, which the Stanford parser renders into the phrase tree (S (VP (VB show) (NP (NN tool) (NN bar)))) or:

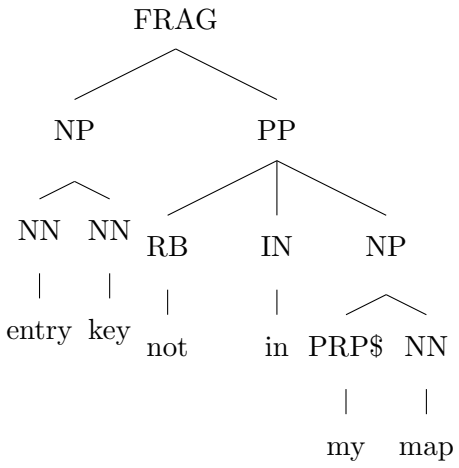


where S represents an imperative statement, NP a noun phrase and VP a verb phrase[5]. Few

[5]A list of Penn Treebank PoS and phrase tags can be found in Appendix E

of the trees returned by the parser contain clausal elements such as S and SINV (representing a subject-auxiliary inversion like "Is the list empty?"), so these are ignored and the top-level phrasal elements treated as summarising the phrasal structure of the name. The example, `showToolBar`, is therefore summarised as a verb phrase.

Names with more complex structures are similarly summarised using the top-level phrases. For example, `entryKeyNotInMyMap` (Polyglot) has the phrase structure tree:



and is summarised as NP PP, where IN is a preposition, PRP$ a possessive personal pronoun, PP a prepositional phrase, and RB an adverb.

6.2.5 Use of Known Abbreviations

Abbreviations of single words are formed by the truncation of words (e.g. `impl` for 'implementation'), and the elision of letters (e.g. `ctxt` for 'context'). Multi-word abbreviations are created by applying one or both processes to two or more words (e.g. `regex` for 'regular expression') (Hill *et al.*, 2008).

RQ 6 asks about the content types that developers use in identifier names. So far I have identified ciphers, type acronyms, redundant prefixes, words, acronyms and abbreviations. The partition U contains abbreviations that are known, such as the examples given above, and many that aren't recognised like `TRGDFTRT` found in Derby. As part of the answer to RQ 6 I distinguish between tokens that can be recognised as abbreviations, even though their accurate expansion might not be trivial. A known or recognised abbreviation is one that is found in the abbreviation dictionaries in `MDSC`, which are formed from the lists of abbreviations

(a) extracted from SCOWL, (b) with known expansions from the AMAP project, and (c) compiled from my observations of names[2].

6.2.6 Threats to Validity

In addition to the PoS tagger model accuracy described above, there are threats to construct and external validity.

Phrase structure grammars allow the recognition of the aggregation of types of words into grammatically coherent groups, but there is no guarantee that the groups are meaningful. Whilst 'The cat sat on the mat', for example, is semantically correct, exchanging the nouns creates an absurdity that is also grammatically correct. Accordingly, there is a threat to *construct validity* from an underlying assumption that the developers of the subject projects have created meaningful rather than absurd names. However, the experimental technique cannot distinguish between the two.

A minor threat to construct validity arises from my choices of acronym, cipher and word lists used to partition declarations into the C and P categories. Those lists may not coincide with the vocabulary used by the developers of all the projects surveyed — particularly the domain-specific terms and acronyms used. Consequently, some names may have been assigned to the U partition, resulting in a reduction of the size of P. A further concern is that EJS specifies the use of the ciphers `x`, `y` and `z` for coordinates. As there is no direct type correspondence, for this survey the definition was widened to include any numeric type.

Another threat results from the order of tests which reflect the working definition of an abbreviation, i.e. any token that is not a cipher, type acronym or English word. Accordingly, abbreviations such as `ID` that are also English words ('id' is a psychoanalytical term) are recognised as words rather than abbreviations, and the name will be put in a potentially different partition from the developer's intended meaning of the token.

Threats to *external validity* arise because the experiment is constrained in two dimensions. First, I analysed only projects developed in Java, prior to v8, to take advantage of its strong typing; and, secondly, I analysed projects where names were constructed using English words. Accordingly I cannot be sure that the findings may be applied to less strongly typed programming languages, or that developers who create names using languages other than English use a similar phrasal structure.

6.3 Results

6.3.1 Name Content Types (RQ 6)

The components of a name are its tokens. Each project's tokens were partitioned into 6 sets (not bags), for each name species. For example, the local variable names in Tomcat are composed of 12 ciphers ($\mathbb{C}$), 121 type acronyms ($\mathbb{T}$), 8 redundant prefixes ($\mathbb{R}$), 1206 English words and known acronyms ($\mathbb{W}$), 104 recognised abbreviations ($\mathbb{A}$) and 163 unknown tokens ($\mathbb{U}$).

Note that whilst sets $\mathbb{W}$, $\mathbb{A}$ and $\mathbb{U}$ are by definition mutually disjoint, the others are not, e.g. some tokens may occur both as cipher and type acronym, or as prefix and unrecognised token (if not in the first position of a name). I define a project's *vocabulary*, for a particular name species, to be the bag of tokens obtained from the multiset union of the 6 sets for that species, e.g. Tomcat's local variable vocabulary consists of 1614 tokens.

Sets $\mathbb{C}$ and $\mathbb{T}$ correspond to removing the duplicate names in C and T, because each cipher and type acronym consists of a single token. The names in P consist only of tokens from $\mathbb{R}$ and $\mathbb{W}$, whilst the names in U have at least one token in $\mathbb{A}$ or $\mathbb{U}$, besides, possibly, others from $\mathbb{R}$ and $\mathbb{W}$.

Besides the unique tokens within each set, I also consider all occurrences of tokens in declarations. For Tomcat, the 12 unique ciphers are declared 1417 times, while the 163 unique unknown tokens occur in 743 declarations. Table 6.5 shows, for each species, the distribution of each type of token as a proportion of a project's vocabulary and, in parentheses, as a proportion of all occurrences.

Table 6.5 shows that at least 61% of a project's reference name vocabulary are words, their use being more common in field names. Acronyms and ciphers are most common as formal argument and local variable names. Recognised abbreviations form at most 10% of a project's vocabulary, and the mean and quartile figures are similar for unrecognised tokens ($\mathbb{U}$), but there are some projects where more than 20% of the vocabulary is unrecognised. Outliers are also found in other types of token. The use of redundant prefixes ($\mathbb{R}$) in JUnit, for instance, is an order of magnitude greater than any other project studied, accounting for 31.5% of all tokens in field names. Similarly, the use of words and acronyms as tokens in formal arguments in Groovy is remarkably low at 20.2% of occurrences, some 36% lower than

Table 6.5: Distribution of proportions of unique tokens within vocabulary and, parenthesised, within all occurrences

		Field	Formal Argument	Local Variable
$\mathbb{C}$	**Minimum**	.000(.000)	.003(.001)	.002(.008)
	1st Quartile	.000(.000)	.009(.034)	.007(.038)
	Median	.002(.000)	.013(.047)	.009(.050)
	Mean	.002(.001)	.015(.049)	.011(.061)
	3rd Quartile	.003(.001)	.019(.064)	.012(.076)
	Maximum	.009(.008)	.049(.159)	.034(.203)
$\mathbb{T}$	**Minimum**	.000(.000)	.013(.002)	.014(.004)
	1st Quartile	.003(.000)	.033(.012)	.049(.024)
	Median	.008(.002)	.052(.023)	.063(.037)
	Mean	.010(.003)	.053(.028)	.066(.042)
	3rd Quartile	.014(.004)	.067(.039)	.084(.051)
	Maximum	.039(.027)	.113(.104)	.123(.157)
$\mathbb{R}$	**Minimum**	.000(.000)	.000(.000)	.001(.000)
	1st Quartile	.003(.002)	.005(.003)	.005(.007)
	Median	.005(.007)	.007(.005)	.006(.010)
	Mean	.005(.021)	.007(.010)	.007(.012)
	3rd Quartile	.006(.023)	.009(.009)	.008(.015)
	Maximum	.016(.315)	.021(.081)	.016(.053)
$\mathbb{W}$	**Minimum**	.674(.633)	.653(.202)	.610(.542)
	1st Quartile	.828(.861)	.763(.768)	.730(.723)
	Median	.864(.891)	.812(.816)	.773(.775)
	Mean	.860(.880)	.806(.801)	.766(.767)
	3rd Quartile	.906(.921)	.850(.860)	.813(.841)
	Maximum	.963(.981)	.924(.975)	.876(.911)
$\mathbb{A}$	**Minimum**	.008(.013)	.024(.010)	.032(.017)
	1st Quartile	.033(.040)	.045(.041)	.052(.057)
	Median	.043(.053)	.059(.063)	.063(.077)
	Mean	.044(.055)	.057(.076)	.063(.081)
	3rd Quartile	.054(.068)	.066(.094)	.074(.096)
	Maximum	.078(.121)	.098(.411)	.100(.393)
$\mathbb{U}$	**Minimum**	.012(.003)	.013(.002)	.028(.008)
	1st Quartile	.038(.017)	.040(.017)	.059(.025)
	Median	.061(.037)	.053(.026)	.082(.032)
	Mean	.080(.040)	.062(.036)	.088(.037)
	3rd Quartile	.096(.050)	.072(.038)	.101(.044)
	Maximum	.281(.162)	.227(.370)	.243(.090)

in any other project.

6.3.2 Phrasal Structures (RQ 7)

There is no reason to tag and parse the names in C and T, and those in U require further processing before they can correctly tagged and parsed. Thus RQ 7 concerns only the 2.6 million declarations in P, with names composed of words, acronyms and redundant prefixes.

Applying the Stanford Parser to PoS tagged field names in P identifies the most common phrasal structures shown in Figure 6.1 where the whiskers extend at most to 1.5 times the interquartile range from the box. Unsurprisingly, given that field names largely represent the attributes of entities and that the JLS and EJS encourage developers to use nouns and noun phrases to name them, the overwhelming majority of field names in P are noun phrases (NP). Redundant prefixes are used in field names in some projects, and these are seen in Figure 6.1 as a redundant phrase followed by a noun phrase (RDP NP). The lowest outlier for NP and the highest for RDP NP is JUnit, where redundant prefixes are used extensively. The fifth category, a noun phrase followed by a verb phrase (NP VP), contains names that might be a sentence, e.g. `FIELD_IS_VOLATILE` (JDK). An alternative NP VP structure is found in *multi-part* or *multiple phrase* names such as `ConfigurationView_replaceWith` (Eclipse). I discuss multi-part names further in Section 6.4. Importantly, Figure 6.1 shows that though noun phrases are the dominant phrasal structure for field names, other phrasal forms also need to be considered by automated analytical techniques.

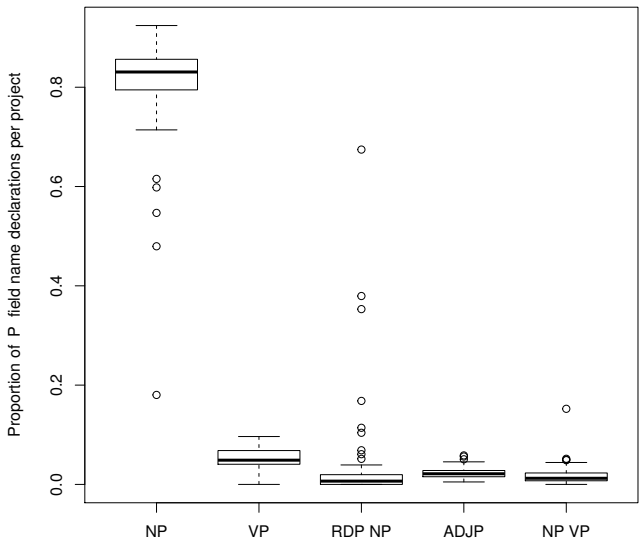


Figure 6.1: Proportions of most common field name phrasal structures in P

Table 6.6 shows the mean proportions of the 5 most common phrasal structures for each species. In this and following tables, proportions are given in parentheses when they are not amongst the five most common. The NP VP pattern seen in field names occurs much less often in formal argument and local variable names. Prepositional phrases (PP) are more common in local variable names, e.g. `beforeMethods` (Stripes), and adverbial phrases such as `forward` (OpenProj declaration `boolean forward`) are found in formal arguments. Summing the figures for field names for NP, VP, RDP NP, ADJP and NP VP shows these phrasal forms are used in 91% of the names in P. The proportion rises to 96% for formal arguments.

Table 6.6: Mean proportion of 5 most common phrasal structures in P

	NP	VP	RDP NP	ADJP	NP VP	PP	ADVP
Field	.804	.051	.039	.023	.019	(.006)	(.005)
Formal Argument	.908	.021	.011	.022	(.000)	(.006)	.008
Local Variable	.884	.023	.012	.023	(.004)	.015	(.011)

Non-boolean reference names are predominately noun phrases (Figure 6.2 and Table 6.7). The outliers found in JUnit in Figure 6.1 are also found in Figure 6.2.

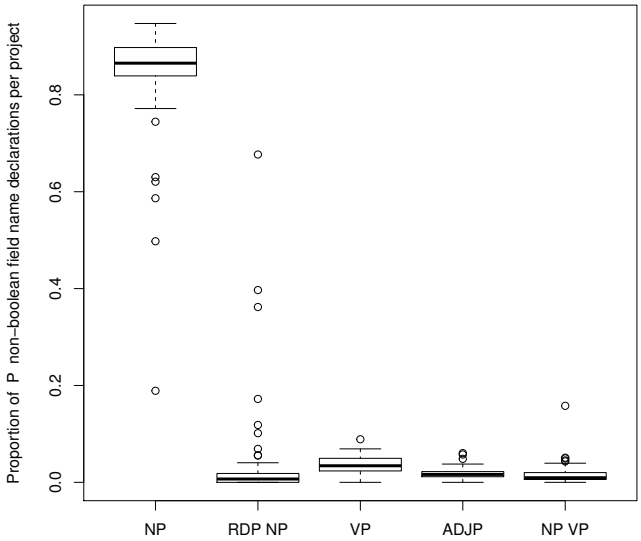


Figure 6.2: Proportions of most common non-boolean field name phrasal structures in P

The distribution of phrases in boolean field names is shown in Figure 6.3. Liblit *et al.*'s

Table 6.7: Mean proportion of 5 most common phrasal structures for non-boolean declarations in P

	NP	RDP NP	VP	NP VP	ADJP	PP	ADVP
Field	.839	.040	.037	.017	.019	(.005)	(.006)
Formal Argument	.935	.011	.008	(.000)	.018	(.005)	.007
Local Variable	.902	.012	.016	(.002)	.021	.015	(.010)

observation that developers use noun phrases for booleans where the verb 'to be' has been elided may be confirmed by the noun phrase being the largest category. Names beginning with a verb have been divided by the Stanford Parser into two categories, those that are verb phrases (VP) and those that are composed of the 3rd person present form of a verb followed by a noun phrase (VBZ NP), which mixes a PoS tag with a phrasal level tag. The apparently multi-phrase form VP NP arises in boolean names (Table 6.8), particularly in local variables, because the Stanford Parser appears to have difficulty parsing some combinations of verbs and nouns. Names like `isShowLines` (JasperReports) are difficult to parse into any form of phrasal structure, because they are not English phrases. These issues are discussed further in Section 6.4.

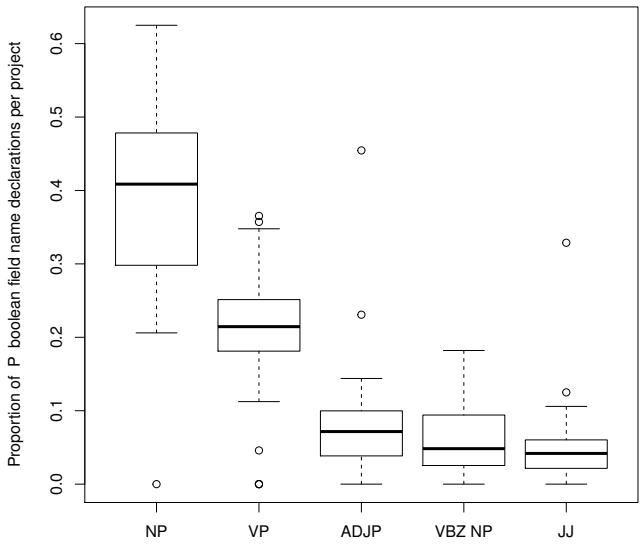


Figure 6.3: Proportions of most common boolean field name phrasal structures in P

In Table 6.8 some formal argument names are categorised as consisting of adjectives,

rather than forming adjectival phrases. The Stanford Parser sometimes categorises names consisting of a single adjective as an adjectival phrase (ADJP) and on other occasions as a sentence fragment containing a single adjective (JJ). Testing shows this behaviour to be consistent, and I have left the results as reported by the parser. Summing the figures in the ADJP and JJ columns gives the extent of the use of adjectival phrases in boolean names.

Table 6.8: Mean proportion of most common phrasal structures of boolean names in P

	NP	VP	ADJP	VBZ NP	VP NP	JJ
Field	.394	.212	.078	.062	(.039)	.044
Formal Argument	.408	.267	.099	.057	(.025)	.052
Local Variable	.431	.201	.075	.072	.049	(.047)

Tables 6.7 and 6.8 answer RQ 7 by showing that, for names composed only of prefixes, words and acronyms (P), developers tend to use the phrasal structures identified by Liblit *et al.*, including the use of nouns or noun phrases as names for booleans, where plausible use of the verb 'to be' has been elided. Table 6.8 shows use of adjectival phrases (ADJP) and bare adjectives (JJ) where, similarly, plausible use of 'to be' has been elided, e.g. `empty` (XOM).

The metaphors proposed by Liblit *et al.* (2006) for names were intended to be an approximation to their phrasal structure, rather than a comprehensive list. Two of the metaphors are relevant to Java reference names: *data are things* (which corresponds to the use of noun phrases) and *true/false data are factual assertions* (corresponding to the indicative mood, e.g. `contains`, `isEmpty`). My results show that some 85% or more of non-boolean field names in P, and similar proportions of formal argument and local variable names, are constructed from noun phrases. The finding shows widespread use of Liblit *et al.*'s metaphors and that developers also use other forms of phrasal structure in names.

For boolean names in P the picture is more complex. Liblit *et al.* (2006) observed that some boolean names are factual statements with the verb 'to be' elided. The latter case includes isolated nouns, or noun phrases and adjectives, e.g. a local variable named `empty` (XOM), which might easily, and more clearly, be named `isEmpty`. The results for boolean names (Table 6.8) show a large proportion of noun phrases and isolated adjectives. Manual

inspection of a sample of isolated adjectives and adjectival phrases confirms Liblit *et al.*'s observation. Many noun phrases could reasonably be preceded by 'is'. Indeed prepositional phrases could also, often, be preceded by 'is'.

From these findings a production for single phrase reference names can be identified: RDP?(NP|VP|ADJP|PP|ADVP). The use of redundant prefixes RDP is optional for all single phrases because while they are not always represented in the 5 most common patterns in Tables 6.6, 6.7 and 6.8, reference names such as `fIsDefaultProposal` (Eclipse) and `bInRefresh` (Vuze) are found in practice. The production differs in two regards from the phrase structure grammar used in SWUM (Hill, 2010): firstly, the production only attempts to describe single phrases, and secondly, as a result of this investigation, includes adjectival and adverbial phrases.

6.4 Discussion

In this section I discuss the results and how they provide opportunities for improving techniques used to analyse names.

6.4.1 Problems for PoS Tagging

There are a number of issues I encountered that are important considerations for the extension of this work or its application in program comprehension tools, and I outline some potential solutions.

The investigations described in this and the preceding chapter were undertaken using the Stanford PoS Tagger using models that I had trained on identifier names. The Stanford PoS Tagger was developed to process sentences, which are generally longer than identifier names and, thus, contain more information to support the tagging process. Consequently, some forms of word use in names may lead to incorrect PoS tagging.

Homographs (words with same spelling but different meaning) can mislead the PoS tagger, e.g. some noun phrases are actually verb phrases where the leading verb has been mis-tagged as a noun or an adjective. Sometimes this is an error on the part of the tagger, on other occasions information in the name is insufficient to differentiate the use of a word such as 'duplicate' as an adjective, noun or a verb in names like `duplicate_peer_checker` (Vuze). Contextual information from the declaration may support a particular tagging.

Developers do not always use English words in expected ways, though they are used in ways that may be straightforward for humans to understand. An example is `AboutDialog` (OpenProj) where 'about', a preposition, is used as a noun and the name is intended to be a noun phrase. A similar, common unconventional use of words is the specialised use of verbs and verb phrases as nouns in the names of GUI elements that represent user activity, e.g. `SaveAllAction` (Eclipse, NetBeans). In prose, it is possible to use devices such as an article before the phrase, hyphenation, and possibly quotation marks to indicate the unusual nature of the word usage. However, the latter two can't be used in Java names, and the use of articles in names is likely to be seen as superfluous by developers.

A heuristic may be used to identify the conditions under which a verb might be used as a noun (a grammatical neologism). However, any heuristic would also need to be able to recognise the use of verbs in reference names. For example, the exit behaviour of the Java Swing top-level window classes is controlled by a group of integer constants, including `CLOSE_ON_EXIT`, where the conventional use of the verb is entirely clear and understandable. Similarly, a style of identifier naming using verb phrases for parenthetic pairs of characters or elements may be found in some text processing code. For example, the Eclipse plugin development environment contains code for writing HTML where string constants such as `OPEN_H4` and `CLOSE_H4` are defined. A further consideration is that non-standard use of a word may occur at a very low frequency and, thus, there is limited evidence to support any decision to treat the word differently.

An alternative approach might be to reclassify verb PoS tags as nouns in the names of specific GUI classes, such as `Action`. More detailed study would be required to identify relevant classes for which such re-tagging would be appropriate. However, a hard coded solution may be less desirable than a reliable heuristic.

For the investigation of reference names reported in this chapter I used the Stanford Parser to analyse the structure of PoS tagged names. The Stanford Parser is a probabilistic parser trained on a corpus of sentences extracted from newspaper articles. While the parser performed well with names, the more technical vocabulary and expressions used in names may result in parse trees that appear to be incorrect. The authors of the parser, however, acknowledge the parser has some limitations when processing sentences[6].

[6] `http://nlp.stanford.edu/software/parser-faq.shtml#i`

Some names that might seem easy for humans to identify as phrases are more difficult for software designed to process sentences to interpret. For example the name `isTopLevelChange` was erroneously identified by the parser as two separate phrases. The issue in this instance is that the parser doesn't see 'top level change' as a noun phrase unless the preceding verb is removed. Through experimentation, I found that inserting the determiner 'a' between the verb and the noun phrase leads the parser to identify the noun phrase as expected.

Another group consists of names composed of English words in non-phrasal combinations, including the nonsensical, such as `ignoreActivate` (Eclipse), `isShowLines` (JasperReports) and `manual_lazy_haves` (Vuze). Names in this group are candidates for refactoring. While it might be straightforward, for example, to refactor `ignoreActivate` to `ignoreActivation`, the refactoring is difficult to justify without prior source code inspection.

The final group of problematic names to consider in W have more than one phrase. Examples include error reporting values and string constants like `ER_CANT_CREATE_URL` where 'ER' is used as a prefix, and name that references internationalised strings and are also used as keys in resource bundles. As keys the names have to contain sufficient information on their purpose to be useful to the reader of the resource bundle. One such name in Eclipse has the name `CompilersPropertyPage_useprojectsettings_label` and some can be extremely long — 39 tokens in one case (Section 6.2.1). In this case typography is used to group the tokens of each constituent part and to separate the three parts of the name, a technique Høst and Østvold (2008) describe as *delimiter precedence*. Concatenating the component parts does not form a single sentence. The concern with this group of names is that mechanisms to parse them using conventional natural language tools may need to make judgements about dividing them into phrases on the basis of typography — which is not used consistently by development teams. In this case, the name represents a string used in the GUI class `CompilersPropertyPage` that is a 'label' widget on the displayed page, and relates to an instruction to use project settings for which the text is available in translation. An alternative solution might be for development teams to use some mechanism to specify their naming conventions so that they might also be used by analytical tools to identify the component parts of the name. This possibility is discussed in more detail in the following chapter (Section 7.6).

A further concern for automated tools is that what appear to be similar phrases, to the

casual reader, are categorised differently by the Stanford Parser. In Figure 6.3 and Table 6.8 results for boolean declarations of the form 'is ...' are reported in two categories VP and VBZ NP. In practice `isEmpty` (Groovy) is categorised as a verb phrase and `isZip` (BCEL) tagged as is/VBZ zip/NN and parsed as the phrase tree (SINV (VBZ is) (NP (NN zip))), where SINV is a subject-auxiliary inversion. The difference arises because the isolated adjective in `isEmpty` is recorded as an adjectival phrase leading to the parse tree (FRAG (VP (VBZ is) (ADJP (JJ empty)))), from which the top-level verb phrase (VP) is extracted to categorise the name. If a noun is added, as in the phrase 'is empty room', 'empty room' is recognised as a noun phrase and the whole phrase is recognised as a subject-auxiliary inversion. A solution is for analytical tools to be aware of the similarity of meaning between the two categories of phrase.

6.4.2 Boolean Names

The limited information available in identifier names makes the task of correctly differentiating between the usage of homographs challenging for the tagger and appears to contribute to the inaccuracies observed when tagging boolean names. The first issue concerns homographs found at the start of a name that can be both a noun and an imperative form of a verb, e.g. *request* or *update*. Consider a name such as `requestValue`: is it a declarative or imperative statement? With only limited information available, the tagger tags 'request' as a noun rather than a verb. This behaviour is quite reasonable, and appears to be a limitation of the PoS tagger in the context of names. The second issue concerns homographs that are the past participle of a verb and an adjective, for example in a phrase such as 'is enabled'. Santorini (1990) provides a series of tests to be applied to whole sentences to distinguish between adjectives and past participles. I have applied Santorini's rules insofar as possible to names in the training and test sets, but the limited information available in names makes it difficult to differentiate between the senses of homographs in automated analysis.

A necessary constraint on this analysis was to rely only on the information contained in the name for phrasal analysis, in order to observe the extent to which developers use Liblit *et al.*'s metaphors. Developers of any practical software engineering tool are also able to leverage the declaration context as a source of information to support phrasal analysis. For example, a word such as `request`, in the absence of corroborating evidence such as a

subsequent determiner, might be tagged as a noun when it is part of a non-boolean name and otherwise tagged as a noun or as a verb, with the phrasal structures resulting from the alternatives used to support a preferred PoS tagging.

6.4.3 Abbreviations and Neologisms

The name partitions C and T, containing ciphers and type name acronyms, are used for generic identifiers and contain no phrasal information. There is little to be gained from expanding ciphers and type name acronyms into words. JLS specifies the use of ciphers and type acronyms for formal argument and local variable names. However, I found declarations of field names in both partitions in some projects. In most cases, fields with generic names are found in classes with coordinates such as `int x` and `int y`, and in inner classes that implement actions such as string processing for the containing class. The use of type acronyms as field names is limited to a few projects. For example, type acronyms used in JBoss appear to have become part of the project vocabulary for some commonly used classes.

Abbreviations used in identifier names vary from the readily expandable `buf` to mnemonics such as `CSTMBCS` (Derby). Abbreviaton expansion techniques have been proposed that use regular expressions to look for possible expansions (Hill *et al.*, 2008), apply speech recognition algorithms (Guerrouj *et al.*, 2012a), and use methods developed for machine translation (Lawrie and Binkley, 2011). A key problem to be solved by approaches to abbreviation expansion is that a given abbreviation may have multiple expansions, and, in some cases, the abbreviation may also be a word, e.g. 'auto', 'in' and 'out' are commonly used in identifier names. Among the names in `INVocD` '*auto*' is used as a contraction of 'automated', 'automatic' and 'automatically' (indeed 'automagic' or one of its forms might also have been meant). Abbreviation expansion might be applied either prior to phrasal analysis or as an iterative solution to identify possible corrections to unanticipated phrasal structures. For example, the boolean name `isAutoActivated` (Eclipse) could be expanded to is/VBZ automated/JJ activated/VBN, is/VBZ automatic/JJ activated/VBN and is/VBZ automatically/RB activated/VBN, allowing the latter to be selected as the candidate expansion.

A lack of abbreviation expansion may result in unintended interpretation of grammatical structure where an abbreviation is also a word. For example, common truncated abbreviations such as `inFile` (NetBeans) and `outFile` (ANT) are PoS tagged as IN NN and thus seen as

prepositional phrases by the Stanford Parser, instead of the expanded noun phrases 'input file' and 'output file'. The context of the declaration including the type name could help differentiate between the intended meaning and contribute to abbreviation expansion.

Neologisms and spelling mistakes are also included in the U bag. Spelling mistakes can be identified and, potentially, corrected using spellchecking software such as `MDSC`. Recognising neologisms is more challenging. Techniques exist to identify neologisms derived from existing words — `INTT`, for example, contains a very simple method for identifying derived neologisms (Section 4.3.4), and techniques have been developed for identifying word blends (Cook and Stevenson, 2010) — but completely new words and 'grammatical neologisms', such as the introduction of *text* as a verb, are less easy to detect (Janssen, 2012).

6.4.4 Future Work

The discussion identifies further work to be undertaken in order to improve the analysis of names and, thereby, software engineering tools that rely on identifier names to support program comprehension, software maintenance and other tasks. In summary, the directions for future work are:

- The improvement of PoS tagging algorithms to develop solutions that can distinguish homographs with different PoS (verb and noun, past participle and adjective, etc.), and identify the use of plurals and possessive nouns where apostrophes have been elided.
- The development of algorithms to recognise and parse names consisting of multiple phrases.
- The application of existing neologism recognition techniques to identifier names.
- The development of heuristics to identify non-phrasal combinations of words.

6.5 Summary

Reference name declarations constitute around 69% of all declarations in source code and are therefore a rich source of information for developers and tools that perform or support software maintenance tasks, including program comprehension and code search. The analysis of reference names in the corpus reported in this chapter contributes:

- the first survey of reference names, including the distribution of their content types (types of tokens) and forms (phrasal structures);
- the empirical confirmation of extensive adherence to forms suggested in the literature, including the use of Liblit *et al.*'s metaphors;
- the identification of other reference name forms;

The analysis consists of a quantitative study of the content types and most frequent forms of 3.5 million declarations of 522,857 unique field, formal argument and local variable names, in the `INVocD` corpus, complemented by an in-depth qualitative observation of individual names, gained from manually tagging almost 46,000 names.

I found that the majority of names use the content types suggested in naming conventions (ciphers, type acronyms, dictionary words, etc.) and consist of phrases, especially phrases that largely follow the grammar identified by Hill and the metaphors observed by Liblit *et al.*, often used by program comprehension tools.

However, I also found a non-negligible number of names containing tokens (e.g. 18% of field names on average) that require further processing (abbreviation expansion or spell and neologism checking), which is a barrier to tool-supported program comprehension. However, there can be considerable variation in the proportions of the categories between projects with, in extreme cases, more than 70% of the formal arguments in some projects requiring further processing.

Moreover, I found that developers use a richer range of phrases than documented in previous work, including long names composed of multiple phrases, adjectival and adverbial phrases, and non-phrasal names consisting of dictionary words. Accordingly tools need to implement a wide range of techniques so that they are capable of processing the variety of tokens found in names.

The findings described in this chapter provide insights into the ways reference names are constructed in practice, and the issues they raise for program comprehension, whether by humans or by software tools that rely on standard techniques. The close inspection of how names are tagged and parsed has led to the identification of several possible avenues of further research and development in the natural language processing of reference names.

In the following chapter I investigate the adherence of reference names in the corpus

to three sets of naming conventions to understand the extent to which developers adhere to naming conventions, including the use of typography, and to gain insight into the conventions used in practice.

Chapter 7

Adherence to Reference Naming Conventions

A significant question arising from the analysis of reference names reported in the preceding chapter concerns whether developers use the wide variety of reference name content and phrasal structures according to naming conventions. The empirical analysis provides a partial answer because I found unanticipated forms of names — ones that do not conform to conventions — including the extremely long, multi-part names used as keys in resource bundles, and the use of type acronyms as field names. In this chapter I investigate the adherence of the name declarations in the `INVocD` corpus to three sets of naming conventions to understand whether developers follow the naming conventions published by Gosling *et al.* (2014) and Vermeulen *et al.* (2000), and the extent to which the phrasal metaphors observed by Liblit *et al.* (2006) are used in expected ways. The PoS tagging and phrasal analysis techniques developed in the preceding two chapters support the investigation, and `MDSC` (Chapter 6) is used to support the discrimination of name content.

Identifier naming conventions, such as EJS and JLS, are intended to support the readability of source code and also provide some cues, such as typography, suggestions for phrasal structure and the use of words and abbreviations that facilitate the extraction of meaningful information by the reader. Where developers follow conventions there is less work required for program comprehension tools to split names and extract semantic content from source code (Hill, 2010; Abebe and Tonella, 2010). Besides naming conventions, tools that extract semantic content also rely on observations of naming practice, including those made by Liblit

et al. (2006), to support their approach to phrasal analysis.

I investigate the adherence to naming conventions for Java references for two important reasons. Firstly, as previously stated, reference names constitute more than half of the unique names and around 70% of the declarations in the corpus, making them a potentially large source of information for program comprehension and software maintenance tools. Secondly, unlike classes (Chapter 5) and methods (Høst and Østvold, 2008) where naming conventions are largely clear and consistent, reference naming conventions often provide conflicting advice (e.g. JLS recommends the use of abbreviations for formal arguments, while EJS advises the use of words), and provide a bewildering choice of content types and forms, including ciphers, type name acronyms, abbreviations and phrases. Consequently, software development and research tools that use the textual and natural language features of identifiers are processing an information source of unpredictable quality.

To be able to check adherence to such diverse naming guidelines I developed a convention checker, because available tools such as CheckStyle (Burn, 2007), Google CodePro AnalytiX[1] and PMD (InfoEther, 2008), can neither specify nor evaluate phrasal structures, and rely on limited mechanisms to evaluate typography. To simplify the checking of different sets of naming conventions, I also defined a domain-specific language to specify naming conventions in a simple and declarative way. The language and the checker, called `Nominal`, were informed by the study of Java reference names reported in the preceding chapter that observed the forms reference names take in practice without *a priori* judgement about which forms are 'right'. In this chapter, I explicitly judge names as conventional or unconventional, depending on their adherence or not to a naming guideline, and I test the use of typographical conventions.

To evaluate the effectiveness and efficiency of the proposed approach, I defined three different sets of comprehensive naming conventions with `Nominal` and checked the adherence of 3.5 million reference name declarations in the corpus against each guideline. The naming conventions are taken from EJS, JLS and studies of Java identifiers in the academic literature (Hill, 2010; Abebe and Tonella, 2010; Liblit *et al.*, 2006) and also my work reported in the preceding chapter.

In developing and applying `Nominal` to check adherence to reference name conventions, I seek to answer the following research questions:

[1] `https://developers.google.com/java-dev-tools/codepro/doc`

RQ 8 To what extent do projects adhere to particular naming conventions or style?; and

RQ 9 Do some naming conventions tend to be broken more frequently than others?

7.1 Related Work

There is limited literature on the holistic evaluation of the application of naming conventions in source code projects. Boogerd and Moonen (2008, 2009) undertook extensive studies of the adherence of source code to the MISRA-C (MIRA Ltd, 2004) coding standards, which include four rules on naming, but not a comprehensive set of naming conventions. Much of the remaining research evaluates the impact of specific aspects of naming conventions, sometimes in combination with other properties of source code, on program comprehension.

The identifier naming conventions proposed by Relf (2004), that formed the basis of the conventions applied in Chapter 2, were evaluated with software developers. Relf found that acceptance of specific conventions was related to the developers' experience, suggesting that understanding of the value of conventions increases with experience. The conventions were subsequently applied in a tool to support the creation of names during maintenance and development exercises. The subjects — 69 undergraduate computing students and 10 professional developers — using the tool tended to respond to advice from the tool and created more higher quality names (according to Relf's conventions) than the control group (Relf, 2005).

Two groups have studied the relationship between program comprehension and the use of abbreviations and words in identifier names. Takang *et al.* (1996) investigated the influence of comments and identifier names containing abbreviations and words on the comprehension of Modula-2 code by undergraduate programmers. Their results showed that a combination of comments and identifier names containing words rather than abbreviations improved program comprehension. However, they also found no statistically significant difference between comprehension of names consisting of abbreviations, and those composed of words.

Lawrie *et al.* (2006) revisited this problem area undertaking a similar experiment on the impact of identifier naming on program comprehension. Underlying their approach to their

experiment was the understanding that comments are not always present in source code, and that developers have to rely on identifier names alone. Experimental subjects were asked to describe the functionality implemented in one of three variants of methods where identifier names were either single letters, abbreviations or composed of words. The study found that subjects were most confident understanding identifiers containing full words. However, there was, often, only a slight improvement over understanding abbreviated identifiers. Single letter identifiers were the least well understood.

A survey of naming practices in three code bases by Liblit *et al.* (2006) identified phrasal metaphors used by developers to create names with particular roles. The code surveyed was written in C, C++, C# and Java and consisted of Gnumeric, the Java v1.3 class library, and Microsoft Windows Server 2003. I investigated the use of the metaphors in reference name declarations in Chapter 6 finding that they are widely used.

CheckStyle (Burn, 2007) and PMD (InfoEther, 2008) are widely used open source tools that check adherence to coding conventions and include some functionality to evaluate adherence to naming conventions. Both are user configurable, with PMD offering a finer-grained specification of names than CheckStyle. However, both rely on limited property checks — number of characters in a name, for instance — and regular expressions that often do little more than check the capitalisation of the first character of the name, and specify prefixes and suffixes. Other static analysis tools such as FindBugs (FindBugs, 2008) also offer some limited checks of name quality, but are not easy to configure.

The application of statistical natural language processing techniques by Allamanis *et al.* (2014) in NATURALIZE, a coding and naming convention recommendation tool, was motivated, in part, by around 25% of code reviews suggesting changes to naming. Allamanis *et al.* observe that naming conventions, such as JLS and EJS, are prescriptive and project or team conventions tend to evolve by consensus during a project's lifetime and are often not codified. NATURALIZE, however, learns a project's coding and naming styles and recommends refactorings that replicate the conventions used. Consequently it cannot check for adherence to particular conventions and is thus at a disadvantage in the early stages of a project, or where conventions are imposed on a project.

7.2 Methodology

I investigate the adherence of the bag of reference name declarations in the `INVocD` corpus to naming conventions, i.e. the corpus of names investigated in the preceding chapter (Section 6.2.1). This analysis differs in that the detail of the declaration is considered, including any modifiers in the declaration and the type of the identifier. Each declaration indicates the need for developers to choose a name — the content types used in its construction and the phrasal structure, if appropriate — and the typography to use. Reusing particular names reflects the preference for certain identifier forms, typographical styles, phrasal structures or metaphors.

7.2.1 Naming Conventions

I selected three sets of naming conventions to test names against. The first two are EJS and JLS. The third I created to test for the conventional use of Liblit *et al.*'s metaphors, and the findings of others (Hill, 2010; Abebe and Tonella, 2010; Binkley *et al.*, 2011) and the findings on content-types and phrasal structure reported in the preceding chapter.

JLS defines both typography — the use of upper and lower case letters and separator characters such as underscores — and content for Java identifier names. The JLS naming conventions remain largely unchanged since initial publication in 1996 and are familiar to most Java developers. JLS defines a simple typography scheme for reference names. Constant field names — those declared with both the `final` and `static` modifiers — are in upper case with the underscore used to separate tokens, e.g. `LEAF_IMAGE` (Google Web Toolkit). All other field names, and formal argument and local variable names begin with a lower case letter, and the first letter of each subsequent word is capitalised (sometimes known as camel case), e.g. `oldValue` (JabRef).

EJS began life as an internal Java style guide at Rogue Wave Software, and reflects the practice of the company's Java developers. EJS consists of a group of general naming conventions (Rules 9–14) that provide general advice on identifier names — that they should be meaningful, for example — and conventions that provide advice on the typography and content of each species of name (Rules 25–31 cover field, formal argument and local variable names). The typography is, with the exception of acronyms, identical to that defined in JLS. While JLS permits a wide range of content in names, EJS expresses a preference for the use

of dictionary words in names, with a few exceptions.

A survey of naming practices by Liblit *et al.* (2006) identified a number of metaphors, phrasal forms that reflect the role of an identifier name. For example, "true/false data are factual assertions". Liblit *et al.*'s observations both increase the variety of phrasal forms expected beyond those specified by EJS and JLS, and require the division of species into boolean and non-boolean subspecies. Liblit *et al.*'s metaphors have been used as a starting point for the extraction of information from names (Hill, 2010; Abebe and Tonella, 2010) and the preceding chapter confirms the metaphors are widely used. Furthermore, recent research (Liblit *et al.*, 2006; Hill, 2010; Abebe and Tonella, 2010; Binkley *et al.*, 2011), as well as my own work (Chapters 5 & 6), shows that developers use a range of phrasal structures wider than specified in EJS and JLS.

I therefore defined a third set of naming conventions that amalgamates the EJS conventions, because of their emphasis on the use of dictionary words, with the observations of phrasal structures found in practice made by Liblit *et al.* (2006), Hill (2010), Abebe and Tonella (2010), Binkley *et al.* (2011), and myself (Chapters 5 & 6). I refer to this set of conventions as AJC, for **a**ggregated **J**ava **c**onventions. By evaluating adherence to AJC I hope to understand the extent to which the observed diversity of phrasal structure is used in accordance with perceived conventions.

7.2.2 Nominal

To support the evaluation of adherence to naming conventions I developed `Nominal`, a freely available Java library that allows the declarative specification of naming conventions[2]. Each set of conventions — EJS, JLS and AJC — is defined in a separate configuration file.

`Nominal` consists of two components: an evaluation engine that determines the adherence of a name to naming conventions, and a configuration language that allows the declaration of rules read by the evaluation engine. The configuration language draws on the cascading style sheet (CSS) syntax.

EJS and JLS specify the identifier species or subspecies (e.g. constant field) to which a convention applies. The `Nominal` configuration language follows this pattern, defining a range of species and subspecies for which rules may be declared. For each species or subspecies a

[2] `https://github.com/sjbutler/nominal`

set of rules may be given that specify the typography, content and other characteristics of a declaration. For example, Figure 7.1 shows the definition of rules for the *local-variable* species and *local-variable-boolean* subspecies for the AJC conventions. The labels outside the blocks are hierarchical with the species name at the left and subspecies names following. The labels form trees with a single species at the top of each tree. In the case of 'field' for example, the immediate children are 'field-constant' and 'field-variable'. To accommodate rules based on Liblit *et al.*'s and my observations, and the EJS conventions, both are further divided into 'action', 'boolean', 'collection', 'other', and 'string'.

```
local-variable {
    first-char: lower;
    body: mixed;
    content: cipher, NP;
}

local-variable-boolean {
    content: cipher,NP,VP,AdjP,AdvP,PP;
}
```

Figure 7.1: `Nominal` rule definitions for AJC showing rule inheritance and overriding

To avoid repetition or verbose rules, `Nominal` follows a simple model of rule inheritance. The rule for 'local-variable-boolean' inherits the typographic rules from 'local-variable'. The *content* field overrides the definition in the parent species and allows the use of noun phrases and verb phrases[3], observed by Liblit *et al.* (2006), and of adjectival, adverbial and prepositional phrases, that I observed (Chapter 6). Ciphers are the only non-phrasal form of names permitted.

`Nominal` also defines some default rules for each species/subspecies hierarchy including the use of separator characters (none), the use of plurals (unspecified) and redundant prefixes (none). Typically, these rules are used in only a few subspecies.

The input to `Nominal` is a name with metadata about the declaration context (including its type, species and any modifiers used in the declaration) and tokenised versions of the name, as well as part of speech (PoS) tags, and its phrasal structure. In this case the input to `Nominal` is provided by `INVocD` and the PoS tagger for reference names and the technique

[3] `Nominal` uses Penn Treebank notation for phrases: see Appendix E

for identifying a name's phrasal structure described in the preceding chapter.

The evaluation engine contains objects that evaluate each rule specified in the configuration file. There is, for example, an object that evaluates the typographical rule for the initial character of a name, and another that evaluates the use of separator characters. The results for each rule are recorded in an information object that annotates the name declaration passed to the evaluation engine. When all the rules have been evaluated, the name declaration object is returned to the caller. The approach allows rules, and aspects of rules, to be evaluated individually using simple approaches, without the potential risks arising from the implementation of a complex rule in a single statement or regular expression.

7.2.3 Threats to Validity

There are at least three threats to *construct validity*. First, while the conventions used are sufficiently generic and well-known that they are likely to have been followed, there may also have been project specific naming conventions in place that have not been captured and thus cannot be tested.

Second, whilst the accuracy of the PoS tagger, trained on 30,000 unique field names from the `INVocD` corpus, is high, it is not perfect.

Third, as stated in the previous chapter (Section 6.2.6), phrase structure grammars are context free and recognise the aggregation of tagged words into grammatically coherent groups, without any guarantee the groups of words is in a meaningful order.

Threats to *external validity* arise, as discussed in the previous chapter (Section 6.2.6) because the analysis is constrained to Java source code and names constructed in English. Consequently, it is unclear whether the findings can be generalised to other programming languages or names in other natural languages.

7.3 Checking Naming Conventions

In this section I explore which naming conventions can be reliably and accurately checked. The intention of the authors of the EJS and JLS was to provide conventions as guidance to developers. Consequently, convention definitions sometimes lack the precision required to make them easy to test. For example, JLS suggests the use of *mnemonic terms* in local variable and formal argument names similar to those used as "parameters to widely used

classes." (Gosling *et al.*, 2014) The few examples given reflect the use of names taken from Java library classes, but they are not the only widely used classes, particularly in teams using specific library APIs, making the distinction between well-known classes and others arbitrary, and, thus, difficult to test. Reliable checking of conventions requires a convention to be defined with a clear statement of which declarations it applies to and readily identifiable boundaries between declarations that conform to the convention and those that do not. Categorising a declaration is straightforward where a convention is applied to a species. However, where the convention is defined for a subspecies, additional distinguishing information is required to categorise a declaration, e.g. identifying a loop control variable declaration can require more information than is contained in the declaration itself.

7.3.1 Name Content Conventions

EJS and JLS specify a range of content for identifier name tokens. There are differences in the token content specified by the two conventions. EJS advocates the use of dictionary words (Rules 9 and 12), for example, while JLS expresses a preference for the use of abbreviations for formal argument and local variable names. Between them, EJS and JLS define five types of identifier name token content:

1. **Cipher**: JLS specifies a largely familiar set of single letter names (Table 7.1) to be used "... for temporary and looping variables, or where a variable holds an undistinguished value of a type."

2. **Type Acronym**: specified in JLS for short-lived formal argument and local variable names, type acronyms are single token names that are acronyms of the declared type, e.g. `StringBuilder sb`.

3. **Acronym**: either a pronounceable acronym such as AWOL, or an initialism like XML.

4. **Word**: a word found in a dictionary.

5. **Abbreviation**: a string of letters and, possibly, digits, that does not match any of the preceding four categories.

Tokens are tested and annotated with one or more objects indicating which of the five categories they belong to. The tests follow the pattern outlined in the previous chapter. The

first test determines whether the name consists of a single token and if that token is a cipher. A recognised cipher is further tested to determine if it is of the correct type. A name that is not a correctly used cipher, if a single token, is then tested to determine if it is a type acronym. For example, applying the JLS conventions, a declaration of `Iterator i` would fail the test for a correctly used cipher (using the list of JLS ciphers in Table 7.1) and would be classified as a type acronym. Single tokens failing the first two tests and each of the tokens found in longer names, are tested using `MDSC` to determine whether they are words, acronyms or abbreviation, or redundant prefixes in the case of the first token. Where a token is not recognised, it is categorised as an *unrecognised* abbreviation.

Table 7.1: JLS ciphers and their corresponding types

Cipher(s)	Type(s)
`b`	`byte`
`c`	`char`
`e`	`Exception`
`d`	`double`
`f`	`float`
`i,j,k`	`int`
`l`	`long`
`o`	`Object`
`s`	`String`
`v`	a value of some type

EJS (Rule 28) states that developers should compile their own list of acceptable ciphers and 'shorthands' and offers a list of suggestions, a few of which (`c`, `e`, `o` and `s`) are also specified by JLS. EJS also states that using the list given in JLS is acceptable. I amalgamate the JLS list of ciphers with the EJS suggestions to enable us to identify commonly used ciphers when testing the EJS conventions, rather than mark all but a few ciphers as incorrect. EJS also suggests the use of the ciphers `x`, `y` and `z` for coordinates. As there is no direct type correspondence I widen the definition of a coordinate to include any numeric type.

Identifier name content is specified in `Nominal` rules by a line starting with the keyword `content` followed by one or more content types, including `cipher`, `type-acronym`, `abbreviation`, and phrasal structure expressed using the Penn Treebank phrase name abbreviations AdjP, AdvP, NP, PP and VP. For example the content of a local variable in JLS is specified as `content: cipher,type-acronym,abbreviation,NP`. The first three content

types are relevant to single token names only, with `abbreviation` indicating that standalone abbreviations such as `buf`, specified in JLS, are accepted. The phrase name abbreviations can be used alone to specify single phrases, or combined to specify more complex phrasal structures. For example, NP VP specifies a noun phrase followed by a verb phrase.

AJC follows the same typography scheme as EJS and divides declarations into finer-grained subspecies than EJS and JLS. Rather than, for example, treating all local variable declarations identically, names declared boolean can have a phrasal structure that includes the use of verb and adjectival phrases (Figure 7.1). AJC also includes subspecies specifying phrasal structures for string constant declarations and references to GUI actions, where I have observed the use of alternative phrasal structures (Chapters 5 & 6).

7.3.2 Typographical Conventions

Typographical conventions are clearly defined in JLS and EJS except for two grey areas: the use of single underscores in constant field names in JLS, which was resolved by implication from the examples, and the use of upper case acronyms in JLS, which is left unchecked. EJS is unequivocal in stating that only the first letter of an acronym is capitalised when appropriate (Rule 13). In contrast, JLS makes no comment on the matter, but quotes examples of camel case names with upper case acronyms, e.g. the method name `toGMTString()`, which would be `toGmtString()` according to EJS.

The rules applied for all three conventions are that names declared as constant fields are upper case with a single underscore character between each token. Other names are composed of mixed case or camel case, that is the first letter is lower case, the initial letter (if there is one) of each subsequent token is capitalised and the remainder are lower case for EJS (for JLS recognised acronyms may also be upper case). Prior to evaluating the typography, each token is categorised as a word, acronym or abbreviation using `MDSC`, allowing the application of different typographical rules for acronyms for EJS and JLS. Testing the typographical conventions for constants and variables is undertaken by checking the appropriate use of underscores, the capitalisation of the first character of the name, and the capitalisation of the remaining tokens.

The typography of individual tokens is tested rather than the entire name for two reasons. First, to evaluate context based typographical rules for acronyms. Second, the typographical

boundaries marked by the developer do not always match the boundaries between tokens, and token boundaries are sometimes omitted, for example `MAXOPEN_DEFAULT` (AspectJ) consists of the tokens {`MAX`,`OPEN`,`DEFAULT`}, so should be `MAX_OPEN_DEFAULT` to comply with the typography rule for constants. This approach offers significant advantages over the regular expressions used by CheckStyle and PMD. Chiefly, checks at the token level determine that tokens have the correct typography, rather than the name having the appearance of being typographically correct, which is what a regular expression can test. `Nominal` also allows the specification of different typographical rules for acronyms from those applied to words.

7.3.3 Reference Naming Conventions Tested

Field Names

JLS conventions for field names are expressed in two `Nominal` rules *field-constant* and *field-variable*. The typography is as described previously. JLS specifies the content of constant fields as one or more words, abbreviations or acronyms, and of an "appropriate" part of speech. Given the specification of more than one token I have assumed that any appropriate phrase is permitted.

The EJS field name conventions are expressed in four subspecies rules: *field-constant*, *field-constant-collection-reference*, *field-variable* and *field-variable-collection-reference*. The collection conventions are explained further below. The typography rules are identical to JLS, with the exception of acronyms in variable names noted above. The content, however, is restricted to noun phrases for constants and variables reflecting EJS's preference for the use of dictionary words (Rule 9) and the specification of nouns/noun phrases (Rule 26).

AJC adds further subspecies to *field-constant* and *field-variable*. *field-constant-string* contains the specification of a 'complex' phrase type in the content rule to include the very long string constants observed in the preceding chapter. A further subspecies with the suffix *-action* is used to isolate references to instances of classes representing GUI actions that are sometimes named with what appear to be verb phrases, e.g. `SaveAction` (FreeMind). GUI actions are defined as declarations of implementations of the Swing `Action` interface and subclasses of `java.util.EventObject`, the superclass of AWT and Swing events.

Formal Argument and Local Variable Names

The typographical rules for formal argument and local variable names are identical for JLS, EJS and AJC, with the exception of acronyms. Again EJS does not allow the use of type acronyms or abbreviations, and limits content to ciphers and noun phrases. AJC, as with variable field declarations, defines subspecies for boolean names and references to GUI actions, and permits the use of a wider range of phrases, though limited to particular subspecies.

Collections

EJS (Rule 27) specifies that the names of collection references should be pluralised. EJS gives the examples of array declarations — including `Customer[] customers` — which are trivial to identify, and the use of collections classes. I constrain EJS's definition of collection class to those classes in the `java.util` package that implement the `Collection` interface. Note that the `Map` classes are not considered collections by this definition. I also apply the rule to collections declared with generic types, which were not part of the Java language when EJS was written. For example, the declaration `List<Customer> customers`.

The idea of pluralising collection names appears reasonable. However, the EJS rule is very narrowly defined permitting only the use of plurals in generic names where the name is the plural of the type. However, I have seen wider use of plurals, such as `List lines` (ANTLR and many others), and in arrays of primitive types, e.g. `String[] lines` (Rapla). To this end I check both the occurrence of EJS pluralisation, and pluralisation of the names of collections and arrays in order to understand the extent to which developers use plurals in declarations. The EJS rule for pluralisation defines the names it applies to in a way that makes it difficult for the current version of `Nominal` to express the rule. Accordingly the test for EJS pluralisation is hardwired in the library. The more generic rule used to evaluate more widespread use of plurals in reference collections is specified in AJC as follows:

```
local-variable-collection-reference {
    content: cipher,NP;
    plural: true;
}
```

Although it is implied that non-collection names should not be plurals, EJS does not make

an explicit statement to that effect. Accordingly, I treat the pluralisation of names of non-collection references as *unspecified* for the EJS conventions. However, in the AJC conventions the rules for all non-collection references are specified as *singular* to help understand whether declarations that might be expected to be singular are being pluralised.

7.3.4 Other Conventions

In addition to testing for the reference name conventions defined in EJS and JLS, each declaration is tested by default for other known conventions such as the use of leading underscores and redundant prefixes. While contrary to the conventions specified by EJS and JLS, it is helpful to understand if non-conformance with EJS and JLS results from developers following other conventions.

Some developers use redundant prefixes in reference names, either indicating the role of the name or the type (Section 6.2.2). Neither EJS, JLS nor AJC specify the use of redundant prefixes, so any prefix use will be seen as unconventional by `Nominal`. Every multi-token name is tested to determine if the first token is a role-based prefix (e.g. `f` for field, `m` for member and `p` for parameter), or a prefix representing a Java primitive type (e.g. `b`). The use of a redundant prefix is recorded, and I also record whether it accurately reflects the expected role or type.

7.3.5 Conventions not Fully Tested

Some rules specified in EJS and JLS cannot be fully tested. EJS Rule 9 specifies the use of 'meaningful names' and is tested partially by ensuring names are constructed of dictionary words, but does not test whether the name is 'meaningful'. Rule 10 suggests the use of 'familiar names', which the examples imply means business domain terms. Rule 10 might be policed in practice with a dictionary of project-approved domain terms, but in a survey of multiple projects in diverse domains the task is not practical. Rule 11 alerts developers to potential design issues by asking them to 'question excessively long names'. Rule 11 is not evaluated because objective criteria are difficult to identify. Rule 14 specifies that names in the same scope should not differ by case alone, and was tested to the extent that at least one of the names would have unconventional typography. Rules 29 and 30 concern the use of the `this` keyword to distinguish field names, and that arguments to constructors and mutator

methods should have the same name as the fields they are assigned to. Neither rule was tested as they apply to the context the name is used in, rather than its structure and content.

JLS lists a number of standard abbreviations for generic formal argument and local variable names, such as `buf` for a buffer. However, only a few illustrations are given as guidance. I test for the use of abbreviations on the assumption that the lists of abbreviations used, taken from identifier names and in dictionaries, reflect the intent of JLS.

To summarise, the majority of conventions specified in JLS can be evaluated accurately as they are stated. The exception is the vague definition of 'mnemonic' terms. EJS is less sharply defined and I needed to make assumptions to define a clear boundary to some rules. Three EJS rules (25, 26, and 31 concerning typography of references, and phrasal content) are implemented as stated. Seven further rules required interpretation or are partially implemented. Four rules were ignored because they require project specific information or relate to the way in which the name is used.

7.4 Adherence to Specific Conventions (RQ 8)

RQ 8 concerns the extent to which projects adhere to each set of naming conventions. I first examine adherence to typographical conventions, before investigating adherence to the conventions on the use of content types. I also evaluate the conventional use of phrasal structures.

7.4.1 Typography

The typographical conventions used in all three sets of naming conventions checked differ only on the typography of acronyms. Across the 60 projects, developers follow conventional typography much of the time (Table 7.2 gives figures for JLS where the typography of acronyms is ignored). Acronyms are found, on average, in 10% of field declarations, 7% of formal arguments and 9% of local variables. Adherence to the EJS scheme of mixed case acronyms is found at high levels in most projects, but a minority of projects use upper case acronyms extensively.

There is some variation in typographical conformance between projects that may indicate the use of project specific conventions or a sub-optimal application of typographical conventions. The lowest conformance with typographical conventions occurs amongst field

declarations. Inspection of collected declarations suggests that a contributing factor may be developers not following the conventions on when to use constant notation, and, perhaps, the declaration of a field having been changed from constant to variable, or vice versa, without the typography being revised. In Beanshell, for example, there are field declarations such as `final static int normal` and `private static Object NOVALUE`.

Table 7.2: Distribution of the percentage of declarations adhering to typography conventions

	Field	**Formal Argument**	**Local Variable**
Minimum	.43	.80	.78
1st Quartile	.74	.95	.92
Median	.82	.97	.96
Mean	.80	.96	.94
3rd Quartile	.88	.99	.97
Maximum	.97	1.0	1.0

Another possibility is that the definition of a constant used by some developers is more nuanced than that given in JLS or EJS. The Google Java Style guide[4] argues that only those reference declarations that hold a single immutable value should have the typography of a constant. For example, while a declaration of an instance of a collection class using the modifiers `static` and `final` is constant in the sense that the reference to the collection does not change, the values or references stored in the collection can change, so the name should not have the typography of a constant. In contrast, an instance of `java.lang.String` declared `static` and `final` is immutable and therefore constant, and should have the appropriate typography.

A source of noise in some projects is the `java.io.Serializable` interface where each implementing class declares and assigns a value to the field `private final static long serialVersionUID`. The typography is not that of a constant and is imposed by the Java library. Indeed, my analysis shows that the Java Library is just above the mean level of adherence to the typographical and content conventions in JLS.

Eclipse appears to have its own field naming conventions, which differ from those tested against. Some constant fields are named using conventional upper case typography (e.g. `VISIBILITY_PREF`). Field declarations that reference strings containing UI messages are not

[4] `https://google.github.io/styleguide/javaguide.html#s5-naming`

declared as constants, and have a structure in which the first element is the name of the class the message originates from followed by either the message, or an explanation and a message (e.g. `LaunchView_Error_1`). In addition, many variable field declarations have redundant prefixes (e.g. `String fPrefillExp`).

The lowest proportion of compliant field declarations is found in MPXJ where variable field declarations have the prefix `m_`. Jetty, the next lowest at 52%, prefixes some field name declarations — both constant and variable — with one or more underscores, e.g. `_dynamic`. The least compliant typography for formal argument declarations is found in Vuze where a leading underscore is used for some declarations to distinguish them from field declarations when used to pass values or references to fields in mutator methods, and use of the C style underscore separator in some formal argument declarations, e.g. `new_value`. Vuze also has the least compliant typography of local variable declarations (78%). As with formal arguments, underscores are often used both as separators and prefixes.

7.4.2 Name Content

Of the conventions tested, JLS specifies the widest range of content types, while EJS is more restrictive. AJC permits a wider range of phrasal content, but only for particular subspecies. Table 7.3 shows the distribution of the proportion of declarations in the projects investigated that conform to the content conventions for each set of conventions tested. In several cases, differences between EJS and AJC are only apparent at 4 or 5 significant figures.

The differences between the parenthesised and unparenthesised figures illustrate the wide differences in the use of redundant prefixes in the projects studied. I discuss this further in Section 7.5. The specified content types for formal arguments and local variables in JLS include type acronyms, which are excluded by EJS and AJC. Table 7.4 shows the distribution of type acronyms in declarations in the 60 projects surveyed. All maximum values occur in the same project, Polyglot, where 5% of fields, 13% of formal arguments and 21% of local variables are type acronyms.

7.4.3 Conventional Usage of Phrases

Table 7.5 shows the distribution of the use of conventional phrases in name declarations with phrasal content (i.e. not ciphers and type acronyms) in the corpus. The declarations surveyed

Table 7.3: Distribution of declarations adhering to content rules of each convention. Parenthesised figures include declarations with redundant prefixes

		Field	Formal Argument	Local Variable
JLS	**Minimum**	.27(.77)	.84(.90)	.87(.90)
	1st Quartile	.83(.87)	.93(.94)	.92(.93)
	Median	.87(.89)	.94(.95)	.92(.94)
	Mean	.85(.89)	.94(.95)	.92(.94)
	3rd Quartile	.90(.91)	.95(.96)	.93(.95)
	Maximum	.94(.96)	.99(.99)	.97(.97)
EJS	**Minimum**	.27(.75)	.79(.80)	.70(.72)
	1st Quartile	.82(.86)	.88(.89)	.84(.86)
	Median	.86(.87)	.90(.91)	.87(.88)
	Mean	.83(.87)	.90(.91)	.86(.88)
	3rd Quartile	.89(.90)	.93(.93)	.88(.90)
	Maximum	.93(.96)	.97(.97)	.96(.96)
AJC	**Minimum**	.27(.76)	.75(.75)	.70(.72)
	1st Quartile	.86(.89)	.85(.85)	.84(.86)
	Median	.90(.91)	.87(.88)	.87(.88)
	Mean	.87(.91)	.87(.88)	.86(.88)
	3rd Quartile	.92(.93)	.90(.90)	.88(.90)
	Maximum	.97(.97)	.97(.99)	.96(.96)

Table 7.4: Distribution of the usage of type acronyms in name declarations

	Field	Formal Parameter	Local Variable
Minimum	.00	.00	.01
1st Quartile	.00	.02	.04
Median	.01	.03	.06
Mean	.01	.04	.06
3rd Quartile	.01	.06	.08
Maximum	.05	.13	.21

include the use of redundant prefixes, which are removed from the name before the remainder is PoS tagged.

There are only minor differences between the distribution of the proportions of conventional phrase use in declarations for each of the three conventions. The phrasal rules for AJC are applied at a finer granularity of subspecies and allow a wider range of phrasal structures than EJS and JLS. Sensitivity to a wider range of phrases makes a difference for some projects, but, generally, the improvement is marginal. This suggests that while developers do use a richer selection of phrases than advocated in EJS and JLS, they do so with a relatively small proportion of declarations.

Table 7.5: Distribution of the proportions of declarations with expected phrasal structures

		Field	Formal Parameter	Local Variable
JLS	**Minimum**	.77	.75	.61
	1st Quartile	.87	.85	.77
	Median	.89	.88	.80
	Mean	.89	.88	.79
	3rd Quartile	.91	.90	.83
	Maximum	.96	.97	.91
EJS	**Minimum**	.76	.72	.60
	1st Quartile	.86	.84	.77
	Median	.87	.88	.80
	Mean	.87	.87	.79
	3rd Quartile	.90	.90	.83
	Maximum	.96	.97	.91
AJC	**Minimum**	.76	.75	.61
	1st Quartile	.89	.85	.77
	Median	.91	.88	.80
	Mean	.91	.88	.79
	3rd Quartile	.93	.90	.83
	Maximum	.97	.97	.91

Plurals following the narrow EJS convention are extremely rare in the corpus. Developers pluralise collection and array references, but not consistently. They also pluralise some references to numeric values and non-collections objects.

In summary, there is no simple answer to RQ 8. There is considerable variation in the extent to which the projects surveyed follow typographical conventions. In particular, developers in some projects appear to have difficulty adhering to the conventions for field declarations. Developers tend to follow the JLS conventions on the content of name declarations rather than EJS, with many projects using type acronyms. The phrasal content of names follows conventions closely in most projects, though local variable declarations are less compliant than field and formal argument declarations.

7.5 Commonly Broken Conventions (RQ 9)

RQ 9 asks whether some naming conventions tend to be broken or ignored more often than others. Inspection of code where conventions are broken identified patterns of alternative typography and use of content types that, in many cases, appear to be systematic and are probably project conventions.

7.5.1 Typography

The use of leading underscores in declarations is found in just under half of the projects investigated. The greatest use of leading underscores is found in field name declarations in Jetty where two underscores are used as a prefix for the names of variable fields declared with the `static` modifier and a single underscore for the names of other variable fields. The conventions are specified in the Jetty coding standard[5].

Some other projects analysed contain names with a mixture of typographical styles. BCEL, for example, contains a mixture of naming conventions in some classes — lower case words separated by underscores and camel case — without a particular scheme prevailing. Without input from the developers it is impossible to understand whether this, apparently inconsistent, mixture of styles is intentional.

7.5.2 Ciphers and Type Acronyms

The intention of both EJS and JLS is that ciphers are used in formal argument and local variable names. I found ciphers are occasionally used in field names, particularly the declarations of field names of inner classes that provide a generic function, such as string processing, for the outer class, and in classes containing coordinates, such as those in graphics applications or libraries. Rather than being a commonly broken convention, it is a convention that appears to be broken when considered justifiable.

Type acronyms are used extensively in some projects. The distributions shown in Table 7.4 show that developers choose to use type acronyms as formal argument and local variable names as suggested by JLS. However, I also found uses of type acronyms in field declarations, as noted in the preceding chapter. Only 6 projects use type acronyms in 2% or more of their field declarations, including ASM and Polyglot where more than 4% of field declarations are type acronyms.

I identified two further unconventional uses of type acronyms. The first is where the acronym refers to a super type, for example the declaration `ServletInputStream is` (Spring). The second is where a type acronym is used as part of a name, for example `StringBuffer filenameSB` (Polyglot).

[5] `https://wiki.eclipse.org/Jetty/Contributor/Coding_Standards`

7.5.3 Redundant Prefixes

As previously stated, redundant prefixes were found in declarations in some of the projects surveyed. Table 7.6 shows that usage is limited, but that redundant prefixes do play a big role in some projects.

Table 7.6: Distribution of the usage of redundant prefixes

	Field	**Formal Argument**	**Local Variable**
Minimum	.00	.00	.00
1st Quartile	.00	.00	.01
Median	.01	.01	.01
Mean	.05	.01	.01
3rd Quartile	.03	.01	.02
Maximum	.73	.09	.06

The version of JUnit investigated uses redundant prefixes to the greatest extent. 73% of JUnit field declarations — almost all the variable fields declared — are prefixed with `f`. (The naming scheme has been revised in a later version and all redundant prefixes have been removed.) Some 38% of field names declared in Xerces-J are also prefixed with `f`. The convention is applied to most non-constant field declarations, but inconsistently[6]. FreeMind also uses redundant prefixes in field name declarations (13%) and formal arguments (9.2%), but rarely in local variables (0.8%).

To summarise, I found that a few projects seem to apply inconsistent typography, while there is evidence that others have adopted conventions in addition to those defined in JLS and EJS, including the use of redundant prefixes and alternative typography. I speculate that, in these cases, the JLS and EJS conventions do not always meet developers' needs. I also found that some developers use type acronyms and ciphers in what appear quite sensible ways, but other than intended by JLS.

7.6 Discussion

In this section I discuss some issues arising from the results and identify opportunities to extend the research, and possible areas of development for `Nominal`.

[6] A Google search for *xerces-j coding conventions* reveals that the matter is discussed by Xerces-J contributors

7.6.1 Naming Conventions

Evaluation of adherence to reference naming shows that developers tend to follow well-known conventions, and, at times, will also develop and use their own conventions. An interpretation of this finding is that existing naming conventions are sufficient for most projects. However, knowing that developers create or adopt additional conventions, does not explain their motivation for doing so. The varied ways in which some projects apply additional typographical markers and redundant prefixes to field names suggests that fields may be too versatile for specific roles to be inferred from usage, and that developers may find clarifying information supports program comprehension. The use of exceptionally long multi-part names to reference translated strings in resource bundles appears to support this conjecture.

Specifying a set of conventions using the language in `Nominal` is trivial, where conventions are published. However, where developers have not published their conventions, as is the case with many of the projects in the corpus, the conventions would need to be recovered from the source code. The language in `Nominal` might be used to express the mined naming conventions and `Nominal` used to determine how closely a project adheres to its own conventions. Furthermore, the recovery of naming conventions from source code may provide insights into the structure of more complex names and support the extraction of semantic information in software engineering tools.

The motivation for developers to choose to use alternative conventions, i.e. the use of typography, content types and phrasal structures, needs to be understood. The conjecture that there may be issues with programming language design, or deficiencies in conventions appears plausible. However, an investigation of developers' concerns and their motivation for adopting additional conventions could inform the development of both programming languages and naming conventions.

7.6.2 `Nominal`

The language in `Nominal` was designed to specify project-wide naming conventions, and was limited to the kind of conventions specified in EJS and JLS. The conventions used in practice can be more complex. Consequently, `Nominal` needs to be further developed to express and check more complex naming conventions. One possibility is to support the specification conventions at package granularity so that names of API classes, methods and public fields,

for example, might be specified differently to names in packages that are not exposed in the API, e.g. to include some sort of branding like `JRBarPlot` (JasperReports). Further avenues to explore include more detailed specification of content types to allow for mixed content types and prefixes and suffixes, for example in the declaration `StringBuffer filenameSB` the name might be specified so that local variables of the `StringBuffer` type are a noun phrase with a `SB` suffix, or more generically as `NP type-acronym` where a possible form of name is a noun phrase followed by a type acronym. Similarly, implementing the specification and testing of type acronyms derived from super classes and super types appears to be justified.

`Nominal` also has the potential to form the basis of a system to revise names so that they conform to the specified conventions. The way in which conventions are evaluated using simple rules mean that implementing typographical changes would be a straightforward transformation to correct each broken rule. Semantic changes are more difficult to implement and, as pointed out by Allamanis *et al.* (2014), would require human intervention before being accepted.

7.6.3 Future Work

In summary, the directions for future work are:

- An empirical study to investigate developers' motivation for using additional naming conventions.
- The development of a tool to mine the naming conventions used in a body of source code and express it as a set of rules.
- Further development of `Nominal` to specify and check more complex naming conventions, such as long multi-part names and those where content types are mixed, and to allow finer-grained specification of names.

7.7 Summary

Declarations that follow naming conventions are more readable and require less processing by tools, thereby helping support program comprehension, developer onboarding and software maintenance. For Java, references constitute around 70% of the name declarations in source

code, and have the most diverse and complex typographical and content conventions, but existing tools mostly check a rather small subset of simple typographical conventions.

The research described in this chapter makes the following contributions:

- a comprehensive survey of the adherence to three sets of naming conventions of reference name declarations in the `INVocD` corpus;
- insight into the use of alternative typographical and content type conventions by some developers;
- `Nominal`, a configuration language to specify naming conventions and a Java library to evaluate their application;
- `Nominal` definitions for two widely known conventions, EJS and JLS.

`Nominal` is open source and its pre-packaged convention definitions aim to help others adopt, adapt or extend `Nominal` for their own tools and projects.

In the evaluation of 3.5 million reference name declarations I found that the median is over 85% of declarations adhering to conventions, but there is considerable variation in the extent to which developers follow the conventions evaluated, 43–97% for field name typography being the extreme. On inspection, I found occasional projects with inconsistent typography, but many where developers applied a wider range of typographical styles, particularly to differentiate the roles of field declarations, than specified by either JLS or EJS. `Nominal` can be used to specify project specific naming conventions providing opportunities for development teams to codify existing naming conventions for both content and typography and add automated convention checking.

Chapter 8

Conclusions and Future Work

The research presented in this dissertation has analysed the content and structure of Java class and reference identifier names with the intention of improving the understanding of both aspects of names for the construction of better software engineering tools to support both the work of empirical software engineers and software developers.

8.1 Revisiting the Aims and Objectives

The aim of the research reported in this dissertation has been to increase understanding of the content type and phrasal structure of class and reference identifier names through improving techniques for the analysis of names.

A pilot study reported in Chapter 2 identified statistical relationships between names contravening validated naming conventions and measures of source code quality. During the study I discovered two issues. Firstly, I identified limitations to the published techniques for tokenising names. In particular there were limitations to techniques for tokenising names with unconventional typographical boundaries and without typographical boundaries that meant tokenisations could be inaccurate, and digits were treated as separate tokens, so, for example, the acronym `MP3` would be split into the tokens `mp` and `3` thereby losing its meaning. Secondly, I found that some names in the projects studied had content and phrasal structures other than those described in the academic and practitioner literature. The two issues are related because without good techniques for tokenising names it is difficult to identify content types and phrasal structures. Furthermore, limitations to tokenisation and the understanding

of the contents of names have implications for the capabilities of software engineering and research tools that process names.

Arising from the difficulties encountered and observations made during the investigation reported in Chapter 2 I developed the hypothesis that *developers use content types, including natural language content, in Java class and reference identifier names in ways that are richer and more varied than those specified in naming conventions, and described in the academic literature.* To investigate the hypothesis I have sought to answer the principal research question:

> **"What types of content and phrasal structure do developers use in Java class and reference names?"**

Before addressing the principal research question, I developed a toolset to support the research. Chapter 3 describes the development of a new source code model to answer RQ 1.

RQ 1 How can a model for source code be created where identifier names and named AST nodes are both first class citizens? Existing source code models treated identifier names as second class citizens which made extracting names from the model a similar process to extracting names from code. I developed a model of source code vocabulary, implemented in a database, that makes identifier names and their constituent tokens available to the user with metadata, and eliminates the overhead of repeated processing of projects to extract identifier names. `INVocD`, the resulting database, contains identifier names extracted from 60 FLOSS Java projects and provides the corpus studied in the remainder of the dissertation.

Chapter 4 focuses on the identification and implementation of improvements to the then published techniques for identifier name tokenisation to answer RQ 2 and RQ 3.

RQ 2 How can more effective mechanisms to tokenise names with ambiguous or no word boundaries be developed? The published method of tokenising ambiguous word boundaries relied on data extracted from a large number of software projects; the data was not published. I developed an approach where names with ambiguous word boundaries are split using both possible word boundaries and the resulting tokenisations tested to determine which contains more recognised words, abbreviations and acronyms. This approach informed the development of improved algorithms to tokenise names that lack word boundaries. Names are searched recursively for dictionary words and plausible tokenisations identified. A scoring

system is then applied to select the 'best' tokenisation.

RQ 3 How can effective mechanisms to tokenise names containing digits be developed? Previously, digits were considered to be separate tokens in names, which can result in the loss of semantic information. I developed a two stage approach that tokenises a name containing one or more digits to determine if a name contains a recognised digit-containing abbreviation or acronym. Where the use of digits is unrecognised a set of heuristics I devised are used to tokenise the name.

To answer the principal research question, I addressed six specific research questions in Chapters 5–7.

RQ 4 What phrasal structures do developers use in class names? Analysis presented in Chapter 5 confirms the approximation used by other researchers that class names are predominantly nouns or nouns phrases. I also identified use of what appear to be verb phrases to name classes that represent events, particularly actions taken by the user in GUIs. I also identified some of the phrasal structures used by developers in the remaining 10% of class names that reflect classes that represent more complex concepts, some exceptions, and classes that represent aspects of functionality rather than an entity. A case study of FreeMind found some less common forms of names that follow what appear to be local naming conventions, other names which might be refactored to more common phrasal forms, and a very small number of names that may indicate the need to refactor the class.

RQ 5 How do developers incorporate super class and interface names in class names? Developers tend to incorporate fragments of the name of an immediate super class or super type in to a class name, most often as the rightmost part of the name, so that the more adjectival part of the name reflects the specialisation of the super class or type. The analysis reported in Chapter 5 finds that other patterns are also used. Classes that both extend a super class and implement one or more interfaces are sometimes named using only fragments of the super class and super type names. A small minority of names place part or all of the super class or super type name at the beginning of the class name.

RQ 6 What content types do developers use to create reference names, and to what extent is each content type used? Classification of content types reported in Chapter 6 showed that a minimum of 61% of unique tokens in the projects investigated are dictionary words, and as much as 96% of tokens in some projects in the corpus. The

variation is reflected in tokens in declarations, particularly formal arguments, where as much as 80% of tokens in some projects are not recognised words or acronyms. The use of ciphers and type acronyms is relatively low. Recognised abbreviations are found in at most 10% of names. Unrecognised tokens, those that require additional processing such as abbreviation expansion, are found in less than 20% of name declarations on average, but can occur in 50–71% in extreme cases.

The consequence is that tools extracting knowledge from reference names need to be flexible so that they can recognise and process a wide variety of content types, and able to adapt to the variation in the use of content types.

RQ 7 What phrasal structures do developers use in reference names, and how are they related to Liblit *et al.*'s metaphors? Investigation of reference names with phrasal content, i.e. those consisting of words and acronyms, found a predominance of noun phrases in non-boolean names. Small proportions of other phrasal forms were also identified. Similarly to class names, verb phrases are mostly used to reflect GUI actions and events. Liblit *et al.* observe that boolean names should be factual assertions such as 'contains' and 'is empty', they also state that many boolean names are noun phrases to which the verb 'to be' could be added, and this is reflected in my results. Again, though, I found additional phrasal forms such as the use of adverbial phrases, which had not been observed previously.

As well as single phrase forms, I found names that were composed of two or more phrases. The very longest examples in the corpus are names used for translated strings in resource bundles for internationalised applications. Shorter examples, typically those with two parts, are often references to error messages.

My findings confirm Liblit *et al.*'s observations, but come with the caveat that the metaphors described by Liblit *et al.* are used in most, but not all names. Again, the implication for tools processing names is the need for a flexible approach that can accommodate the less common forms of name.

RQ 8 To what extent do projects adhere to particular naming conventions or style? The investigation of adherence of reference names to naming conventions found developers generally apply typographical rules diligently, though, on average, apply the typographical rules for field names less than those for formal arguments and local variables. Additionally, developers on a few projects use well-known typographical conventions much

less, especially in the case of field names. The use of content types, similarly, generally accords with convention, though there are some projects that comply less. A similar picture emerges for phrasal structure, though the least compliant species are local variables.

RQ 9 Do some naming conventions tend to be broken more frequently than others? A few projects in the corpus projects follow particular typographical schemes that, mainly, distinguish variable field names, or between static and non-static variable field names. Some projects follow the typographical convention used in C of using underscores as separators for lower case names, and one project contains a mixture of typographical styles. Redundant prefixes are used extensively in very few projects, where they are found mainly in field and formal argument names. However, they are found in names, including local variables, in most of the projects in the corpus. With a few exceptions where conventions appear to be applied in a haphazard manner, the use of alternative conventions for typography and redundant prefixes, appears to be mostly deliberate. The use of a wider range of phrasal structures — i.e. those not specified in EJS and JLS — is confined to a small proportion of names which seem to be used either to emphasise a specific problem solution, or in a project specific manner, such as the very long names used for internationalised strings.

8.2 Summary of Contributions

The research reported in this dissertation makes contributions in the following areas:

- **Relationships between name quality and source code quality** The pilot study reported in Chapter 2 identified for the first time a correlation between a measure of identifier name quality and measures of source code quality.

- **Identifier Name Tokenisation** Chapter 4 describes a novel technique for the tokenisation of names containing digits, and improved methods for tokenising single case strings.

- **Content Types and Phrasal Structure** The analysis of class and reference names described in Chapters 5 and 6 provides evidence that confirms the common approximations of phrasal structures used in identifier names, and identifies the use of content types and phrasal structures not recorded in the literature. A further contribution is

made by the novel technique of training models for identifier names and using them with an existing PoS tagger.

- **Naming Conventions** Analysis of adherence to naming conventions reported in Chapter 7 identifies the extent to which well-known naming conventions are used in practice. In some projects with reduced adherence to the conventions tested, additional conventions appear to have been adopted to distinguish field names either from other species, or to differentiate particular roles of field names.

- **Tools and Data** A further contribution is made by the publication of the tools developed to support this research and the corpus in the form of the `INVocD` database. Full details are in Appendix D.

8.3 Future Work

Some future work has already been outlined in Sections 4.5.3, 5.4.1, 6.4.4 and 7.6.3. In this section, I summarise the suggestions already made for future work and identify additional topics.

8.3.1 Name Token Content Types

A recurrent theme of this research has been the discovery of novel content types in identifier names, some entirely unanticipated. Naming conventions specify a number of content types to use in names, and these were identified in Chapters 5 to 7. JLS also identify content types they think should not be used in identifier names, but are found in practice, including top-level domain (TLD) names such as com and org, and ISO3166 country codes, which may also form part of TLD names. I also found, for example, hexadecimal numbers in names, ISO and RFC standards designations, and Java bytecode mnemonics. A detailed and extensive survey of a large number of projects — at least an order of magnitude greater than the size of the `INVocD` corpus — analysing and categorising the content types used in names would provide a considerably more comprehensive understanding of the raw materials developers use to construct names, and support the further development of analytical techniques for names.

8.3.2 Identifier Name Tokenisation

The tokenisation of identifier names is non-trivial. My work described in Chapter 4 has built on the work of others, improving on methods to tokenise single case strings, and developing novel methods to tokenise strings containing digits. The techniques rely on a combination of typography and heuristics to identify word boundaries and are effective, but have limitations when processing identifier names constructed with abbreviations — particularly idiosyncratic contractions — neologisms and digits. It is likely that no identifier name tokenisation system can be 100% accurate when faced with real world names. However, it is clear from Hill *et al.*'s evaluation of the available tokenisers (Hill *et al.*, 2013) there remains much room for improvement.

The majority of identifier names are relatively straightforward to split accurately. The minority, however, require additional attention. Sometimes this is a matter of recognising the presence of ambiguous boundaries and applying a number of heuristics to determine the optimal split. In other cases much more intensive techniques are required. The analytical work described in this dissertation, and in the work of others, provides information that can be applied to support the creation of heuristics and new techniques. Some have suggested standardising identifier names in an attempt to constrain the problem space. However, not only is this likely to prove impractical to apply across the software industry, any solution would be unlikely to be applied retrospectively, thus leaving existing and legacy source code without up to date analytical tools.

There are two directions of research that could help improve the accuracy of name tokenisation. The first direction is to improve understanding of how developers construct identifier names, as suggested above, so that content types used can be more readily distinguished and their boundaries identified where they do not match the typography. The second direction of research is to investigate the application of hybrid approaches to tokenisation, such as using knowledge of the anticipated phrasal structure of the names to identify a preferred tokenisation.

Future studies of the development of novel tokenisation techniques should also create reference sets of identifier names and tokenisations that reflect the challenges to tokenisation to support comparative evaluation of tokenisers. A weakness of the research in this area, including my own, is that the test sets of identifier names and their tokenisations tend to

reflect the distribution of problematic names in the population of identifier names, rather than consisting of names with the problems that the tokenisers are trying to solve. Consequently, any test set should consist of a population of at least 1,000 identifier names with each of the typographical or other characteristic that makes tokenisation a challenge. Furthermore, the reference set should be published so that others can evaluate their approach against it to demonstrate any advantages or deficiencies.

8.3.3 Inheritance Trees

In Chapter 5 I investigated two aspects of class naming: the patterns of PoS in names, and the way in which components of super class and super type names were incorporated in class names. The study of inheritance within a single generation demonstrated the characteristics of the way developers incorporate elements of names in the names of subclasses, and raised questions about the reasons developers reuse elements of names, and the patterns of inheritance used. Further investigation of the reuse of name components in inheritance trees could lead to deeper understanding of why particular name elements are reused, and under what circumstances. Any investigation would need to be sensitive to changes to the Java language, especially the introduction of methods to interface definitions in Java v8, which allows the use of mixin classes in Java for the first time.

8.3.4 Neologisms

The use of neologisms in identifier names has been identified by Hill (2010), Gupta *et al.* (2013), and myself (Chapter 4), and we have all tried to make provision in analytical tools for the detection of a particular type of neologism. Neologisms fall into a number of groups, only some of which can be identified programatically. Completely new words, for instance, cannot be identified easily, particularly in an environment such as identifier names where novel abbreviations and acronyms are also found. Hill, Gupta *et al.* and I adopted an approach to identifying neologisms derived from existing or donor words by adding one or more affixes, for example ‘throwable’, which is found in the Java library. The technique is relatively simple to implement, and not particularly expensive computationally. However, derived neologisms can also be created by stemming a word before adding a suffix, e.g. ‘precisify’ (used as a method name in BlueJ). Techniques for identifying this type of neologism have been developed and

could be adapted to work with identifier name processing software.

Word blends such as `grandcestor` (NetBeans) are another type of neologism found in identifier names. Work in the natural language community on the detection of word blends by Cook and Stevenson (2010) suggests that the identification of blends is computationally expensive, and only moderately accurate. Cook and Stevenson's work, however, focuses on the identification of the source of word blends, and it is possible that their method could be adapted to indicate that a string is a plausible neologism, rather than trying to identify the blended words.

8.3.5 PoS Tagging

In Chapters 5 and 6 I report on the novel technique of training PoS tagger models for identifier names. This is just one of a number of approaches to PoS tagging names. Some approaches have used PoS taggers designed to process sentences on names, which are considerably shorter and contain less information than sentences. Using conventional PoS taggers, researchers have generally reported tagging accuracy for names of around 85%. Two research groups have created their own PoS taggers designed specifically to tag identifier names and achieved accuracies between 87% and 97%. In both cases, the tagger undertakes some semantic parsing to identify type names, so that they can be treated as a separate PoS. For research to continue in this area and for the development of practical tools to process identifier names, further exploration of PoS tagging methods is required, and the development of identifier name specific PoS taggers is necessary given the mixture of phrasal and non-phrasal content found in names. An interesting corollary is that any PoS tagger should also be capable of distinguishing nonsensical combinations of words from phrasal combinations.

8.3.6 Naming Convention Specification and Testing

A finding of the evaluation of adherence to naming conventions in the projects in the corpus was that developers, in some projects, adopt or devise additional conventions for field names. Some conventions are adopted to distinguish field names, for example the addition of leading underscores to the name, while others appear to have developed to express specific roles for field names such as keys for translated strings in resource bundles and error messages. Two questions arise that address developers' motivation for adopting additional conventions.

Firstly, have programming language designers created a versatile species in the field name that leaves developers with a need to indicate specific roles? And, secondly, do developers find that using additional naming conventions for field names supports program comprehension?

Chapter 7 introduces `Nominal`, which includes a language for specifying naming conventions. The approach demonstrates advantages for specifying and testing naming conventions over existing tools, and the techniques could be incorporated into those tools to improve them. Furthermore, `Nominal` might be used to express and test naming conventions mined from source code, where the conventions used are not published. However, the expressiveness of the language could be improved to specify a wider range of content types and to allow the specification of more complex identifier name structures mixing, for example, phrasal elements and non-phrasal content types and, thus, better supporting testing of less common conventions.

`Nominal` also provides the opportunity to revisit the pilot study (Chapter 2) with a far more detailed means of evaluating name quality.

8.4 Personal Reflection

Conducting doctoral research is necessarily a challenging exercise and with hindsight I recognise where I might have benefited from different approaches to research.

I was a part-time external research student living far enough from the University that being on campus regularly to engage with academic life within the Department was difficult. While there was the advantage of fewer distractions, the corollary was reduced opportunities for academic stimulation and extended contact with fellow students. The remote researcher should pay close attention to the opportunities that are available both on campus and locally.

My enthusiasm for the topic helped maintain progress. However, the open-ended nature of research process means enthusiasm can sometimes become an obstacle. I found it all too easy to become absorbed in aspects of the topic — particularly analytical problems — that I found very interesting and, consequently, be distracted from the goal of the research project. As I was wisely told early on in my research, knowing when you've done enough is an important skill; acquiring that skill is another matter entirely.

Empirical software engineering research, such as that described in this dissertation, tries to understand the working practices of developers. The core of my research was the investigation

of evidence of naming practices left by developers in source code. In this case the research was conducted by developing and using investigative tools, and has the secondary benefit that the techniques developed might be used to support the work of developers. Working directly with developers proved to be challenging and something that I was only able to do in relatively small ways. Developer documentation and mailing lists for the projects investigated were useful sources of information about specific naming practices, and sometimes offered insight into unconventional names. Informal conversations with developers were always informative, particularly about the day to day problems professional developers face with names, which provided insights into some of my findings. In hindsight, I would have done much more to form relationships with developers and encourage other empirical software engineering researchers who are not working directly with developers to find ways to establish relationships with developers.

Finally, there is the skill that I would dearly loved to have had and that would have made the research process considerably easier: the foresight to understand what notes about my work my future self would need.

8.5 Conclusion

Returning to my hypothesis that *developers use content types, including natural language content, in Java class and reference identifier names in ways that are richer and more varied than those specified in naming conventions and the academic literature*: the research described in this dissertation has, through the development of improved analytical techniques and detailed analysis, addressed the hypothesis showing it to be correct. The research has also identified some of the additional content types and phrasal structures used in class and reference identifier names found in practice, and suggests avenues of research to be pursued to create a more comprehensive understanding of content types. Implementations of the techniques developed to support the analysis are available and are a starting point for the development of improved tools for both researchers working with identifier names and practical software engineering tools to support the work of developers.

Bibliography

S. Abebe and P. Tonella (2010) Natural language parsing of program element names for concept extraction. In *18th Int'l Conf. on Program Comprehension*, Minho, Portugal, pp. 156–159. IEEE.

S. L. Abebe; S. Haiduc; P. Tonella; and A. Marcus (2009) Lexicon bad smells in software. In *Proc. Working Conf. on Reverse Engineering*, Lille, France, pp. 95–99. IEEE.

S. L. Abebe and P. Tonella (2011) Towards the extraction of domain concepts from the identifiers. In *Proc. of the 18th Working Conf. on Reverse Engineering*, Limerick, Ireland, pp. 77–86. IEEE Computer Society.

M. Allamanis; E. T. Barr; C. Bird; and C. Sutton (2014) Learning natural coding conventions. In *Proc. of the 22nd ACM SIGSOFT Int'l Symposium on Foundations of Software Engineering*, Hong Kong, China, FSE 2014, pp. 281–293. ACM, New York, NY, USA. URL `http://doi.acm.org/10.1145/2635868.2635883`.

G. Antoniol; G. Canfora; G. Casazza; A. De Lucia; and E. Merlo (2002) Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983.

G. Antoniol; Y.-G. Gueheneuc; E. Merlo; and P. Tonella (2007) Mining the lexicon used by programmers during sofware [sic] evolution. In *Proc. of Int'l Conf. on Software Maintenance*, Paris, France, pp. 14–23. IEEE.

Apache Software Foundation (2008) Jakarta Cactus - coding conventions. `http://jakarta.apache.org/cactus/participating/coding_conventions.html`.

K. Atkinson (2004) SCOWL readme. `http://wordlist.sourceforge.net/scowl-readme`.

N. Ayewah; W. Pugh; J. D. Morgenthaler; J. Penix; and Y. Zhou (2007) Evaluating static analysis defect warnings on production software. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, San Diego, California, pp. 1–8. ACM.

K. Beck (2008) *Implementation Patterns*. Addison–Wesley.

D. Binkley; M. Hearn; and D. Lawrie (2011) Improving identifier informativeness using part of speech information. In *Proc. of the Working Conf. on Mining Software Repositories*, Honolulu, Hawaii, pp. 203–206. ACM.

D. Binkley; D. Lawrie; L. Pollock; E. Hill; and K. Vijay-Shanker (2013) A dataset for evaluating identifier splitters. In *10th IEEE Working Conference on Mining Software Repositories*, San Francisco, California, USA, pp. 401–404. IEEE.

C. Boogerd and L. Moonen (2008) Assessing the value of coding standards: An emprical study. In *Proc. Int'l Conf. on Software Maintenance*, Beijing, China, pp. 277–286. IEEE.

C. Boogerd and L. Moonen (2009) Evaluating the relation between coding standard violations and faults within and across software versions. In *Proc. of the Int'l Working Conf. on Mining Software Repositories*, Vancouver, Canada, pp. 41–50. IEEE.

J. Börstler; M. E. Caspersen; and M. Nordström (2015) Beauty and the beast: on the readability of object-oriented example programs. *Software Quality Journal*. Early access.

O. Burn (2007) Checkstyle. `http://checkstyle.sourceforge.net/`.

R. P. Buse and W. R. Weimer (2008) A metric for software readability. In *Proc. Int'l Symp. on Software Testing and Analysis*, Seattle, Washington, USA, pp. 121–130. ACM.

S. Butler; M. Wermelinger; and Y. Yu (2015a) Investigating naming convention adherence in Java references. In *Proc. of the 31st Int'l Conf. on Software Maintenance and Evolution*, Bremen, Germany, pp. 41–50. IEEE.

S. Butler; M. Wermelinger; and Y. Yu (2015b) A survey of the forms of Java reference names. In *Proc. of the 23rd Int'l Conf. on Program Comprehension*, Florence, Italy, pp. 196–206. IEEE.

S. Butler; M. Wermelinger; Y. Yu; and H. Sharp (2009) Relating identifier naming flaws and code quality: an empirical study. In *Proc. of the Working Conf. on Reverse Engineering*, Lille, France, pp. 31–35. IEEE Computer Society.

S. Butler; M. Wermelinger; Y. Yu; and H. Sharp (2010) Exploring the influence of identifier names on code quality: an empirical study. In *Proc. of the 14th European Conf. on Software Maintenance and Reengineering*, Madrid, Spain, pp. 159–168. IEEE Computer Society.

S. Butler; M. Wermelinger; Y. Yu; and H. Sharp (2011a) Improving the tokenisation of identifier names. In M. Mezini (ed.), *25th European Conf. on Object-Oriented Programming*, Lancaster, UK, volume 6813 of *Lecture Notes in Computer Science*, pp. 130–154. Springer Berlin/Heidelberg.

S. Butler; M. Wermelinger; Y. Yu; and H. Sharp (2011b) Mining Java class naming conventions. In *Proc. of the 27th IEEE Int'l Conf. on Software Maintenance*, Williamsburg, Virginia, USA, pp. 93–102. IEEE.

S. Butler; M. Wermelinger; Y. Yu; and H. Sharp (2013) INVocD: Identifier name vocabulary dataset. In *Proc. of the 10th Working Conf. on Mining Software Repositories*, San Francisco, California, USA, pp. 405–408. IEEE.

B. Caprile and P. Tonella (1999) Nomen est omen: analyzing the language of function identifiers. In *Proc. Sixth Working Conf. on Reverse Engineering*, Atlanta, Georgia, USA, pp. 112–122. IEEE.

B. Caprile and P. Tonella (2000) Restructuring program identifier names. In *Proc. Int'l Conf. on Software Maintenance*, San Jose, California, USA, pp. 97–107. IEEE.

M. L. Collard; M. J. Decker; and J. I. Maletic (2011) Lightweight transformation and fact extraction with the srcML toolkit. In *11th IEEE Working Conf. on Source Code Analysis and Manipulation*, Williamsburg, Virginia, USA, pp. 173–184. IEEE Computer Society.

P. Cook and S. Stevenson (2010) Automatically identifying the source words of lexical blends in English. *Computational Linguistics*, 36(1):129–149.

A. Corazza; S. D. Martino; and V. Maggio (2012) LINSEN: an efficient approach to split

identifiers and expand abbreviations. In *28th Int'l Conf. on Software Maintenance*, Riva del Garda, Italy, pp. 233–242. IEEE.

M. J. Crawley (2005) *Statistics: an introduction using R.* John Wiley.

F. Deißenböck and M. Pizka (2006) Concise and consistent naming. *Software Quality Journal*, 14(3):261–282.

T. Dilshener and M. Wermelinger (2011) Relating developers' concepts and artefact vocabulary in a financial software module. In *Proc. of the 27th IEEE Int'l Conf. on Software Maintenance*, Williamsburg, Virginia, USA, pp. 412–417. IEEE.

S. Ducasse; N. Anquetil; U. Bhatti; A. C. Hora; J. Laval; and T. Girba (2011) MSE and FAMIX 3.0: an interexchange format and source code model family. Technical report, INRIA. URL `http://hal.archives-ouvertes.fr/index.php?halsid=bvn39r6d5ucvb6e95asroiipj6&view_this_doc=hal-00646884&version=1`.

E. Enslen; E. Hill; L. Pollock; and K. Vijay-Shanker (2009) Mining source code to automatically split identifiers for software analysis. In *6th IEEE International Working Conference on Mining Software Repositories*, Vancouver, Canada, pp. 71–80. IEEE.

J.-R. Falleri; M. Huchard; M. Lafourcade; C. Nebut; V. Prince; and M. Dao (2010) Automatic extraction of a WordNet-like identifier network from software. In *18th Int'l Conf. on Program Comprehension*, Minho, Portugal, pp. 4–13. IEEE.

H. Feild; D. Lawrie; and D. Binkley (2006) An empirical comparison of techniques for extracting concept abbreviations from identifiers. In *Proc. of IASTED Int'l Conf. on Software Engineering and Applications*, Dallas, Texas, USA. IASTED.

C. Fellbaum (ed.) (1998) *WordNet: an electronic lexical database.* Bradford Books.

FindBugs (2008) Find Bugs in Java programs. `http://findbugs.sourceforge.net/`.

R. Flesch (1948) A new readability yardstick. *Journal of Applied Psychology*, 32(3):p221–233.

M. Fowler; K. Beck; J. Brant; W. Opdyke; and D. Roberts (1999) *Refactoring: improving the design of existing code.* Addison-Wesley.

E. Gamma; R. Helm; R. Johnson; and J. Vlissides (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

J. Y. Gil and I. Maman (2005) Micro patterns in Java code. In *Proc. ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, San Diego, California, USA, pp. 97–116. ACM.

J. Gosling; B. Joy; G. Steele; G. Bracha; and A. Buckley (2014) *The Java Language Specification (Java SE 8 edition)*. Oracle, Java SE 8 edition.

L. Guerrouj; M. Di Penta; G. Antoniol; and Y.-G. Guéhéneuc (2012a) TIDIER: an identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 25(6):575–599.

L. Guerrouj; P. Galinier; Y.-G. Gueheneuc; G. Antoniol; and M. Di Penta (2012b) TRIS: A fast and accurate identifiers splitting and expansion algorithm. In *Proc. of the 19th Working Conference on Reverse Engineering*, Kingston, Ontario, Canada, pp. 103 –112. IEEE.

S. Gupta; S. Malik; L. Pollock; and K. Vijay-Shanker (2013) Part-of-speech tagging of program identifiers for improved text-based software engineering tool. In *Proc. of the 21st Int'l Conf. on Program Comprehension*, San Francisco, California, USA, pp. 3–12. IEEE.

M. H. Halstead (1977) *Elements of Software Science*. Elsevier.

M. Heller and C. Simonyi (1991) The Hungarian revolution. *BYTE*, 16(8):131–138.

E. Hill (2010) *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration*. Ph.D. thesis, The University of Delaware.

E. Hill; D. Binkley; D. Lawrie; L. Pollock; and K. Vijay-Shanker (2013) An empirical study of identifier splitting techniques. *Empirical Software Engineering*, 19(6):1754–1780.

E. Hill; Z. P. Fry; H. Boyd; G. Sridhara; Y. Novikova; L. Pollock; and K. Vijay-Shanker (2008) AMAP: Automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proc. of the 5th Int'l Working Conf. on Mining Software Repositories.*, Antwerp, Belgium, pp. 79–88. ACM.

E. W. Høst and B. M. Østvold (2007) The programmer's lexicon, volume 1: the verbs. In *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation*, Paris, France, pp. 193–202. IEEE.

E. W. Høst and B. M. Østvold (2008) The Java programmer's phrase book. In *Proc. of the 1st Int'l Conf. on Software Language Engineering*, Toulouse, France, volume 5452 of *LNCS*, pp. 322–341. Springer.

E. W. Høst and B. M. Østvold (2009) Debugging method names. In *Proc. of the 23rd European Conf. on Object-Oriented Programming*, Genova, Italy, pp. 294–317. Springer-Verlag.

InfoEther (2008) PMD. `http://pmd.sourceforge.net/`.

M. Janssen (2012) Neotag: a pos tagger for grammatical neologism detection. In *Proc. of the Eighth Int'l Conf. on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, pp. 2118–2124. European Language Resources Association (ELRA).

D. Klein and C. D. Manning (2002) Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems 15*, Vancouver, British Columbia, Canada, pp. 3–10.

A. Kuhn; S. Ducasse; and T. Gírba (2007) Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243.

D. Lawrie and D. Binkley (2011) Expanding identifiers to normalize source code vocabulary. In *Proc. of the 27th Int'l Conf. on Software Maintenance*, Williamsburg, Virginia, USA, pp. 113–122. IEEE.

D. Lawrie; D. Binkley; and C. Morrell (2010) Normalizing source code vocabulary. In *Proc. of the International Working Conference on Reverse Engineering*, Beverly, Massachusetts, USA, pp. 3–12. IEEE Computer Society.

D. Lawrie; H. Feild; and D. Binkley (2007a) An empirical study of rules for well-formed identifiers. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):205–229.

D. Lawrie; H. Feild; and D. Binkley (2007b) Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering*, 12(4):359–388.

D. Lawrie; C. Morrell; H. Feild; and D. Binkley (2006) What's in a name? A study of identifiers. In *14th IEEE Int'l Conf. on Program Comprehension*, Athens, Greece, pp. 3–12. IEEE.

T. Lethbridge; S. Tichelaar; and E. Plödereder (2004) The Dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94(0):7–18.

V. I. Levenshtein (1966) Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10(8):707–710.

B. Liblit; A. Begel; and E. Sweetser (2006) Cognitive perspectives on the role of naming in computer programs. In *Proc. 18th Annual Psychology of Programming Workshop*, Brighton, UK. Psychology of Programming Interest Group.

H. Ma; R. Amor; and E. Tempero (2008) Indexing the Java API using source code. In *19th Australian Software Engineering Conference*, Perth, Australia, pp. 451–460.

N. Madani; L. Guerrouj; M. D. Penta; Y.-G. Guéhéneuc; and G. Antoniol (2010) Recognizing words from source code identifiers using speech recognition techniques. In *Proc. of the Conf. on Software Maintenance and Reengineering*, Madrid, Spain, pp. 69–78. IEEE.

A. Marcus and D. Poshyvanyk (2005) The conceptual cohesion of classes. In *Proc, of Int'l Conf. on Software Maintenance*, Budapest, Hungary, pp. 133–142. IEEE CS.

A. Marcus; D. Poshyvanyk; and R. Ferenc (2008) Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300.

A. Marcus; V. Rajlich; J. Buchta; M. Petrenko; and A. Sergeyev (2005) Static techniques for concept location in object-oriented code. In *Proc. 13th Int'l Workshop on Program Comprehension*, St Louis, Missouri, USA, pp. 33–42. IEEE.

T. J. McCabe (1976) A complexity measure. *Transactions on Software Engineering*, SE-2(4):308–320.

S. McConnell (2004) *Code Complete: A practical handbook of software construction.* Microsoft Press, 2nd edition.

MIRA Ltd (2004) MISRA-C:2004 Guidelines for the use of the C language in critical systems. URL `www.misra.org.uk`.

N. Moha; Y.-G. Guéhéneuc; L. Duchien; and A.-F. Le Meur (2010) Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

J. C. Munson (2003) *Software Engineering Measurement.* Auerbach.

J. Nonnen and P. Imhoff (2012) Identifying knowledge divergence by vocabulary monitoring in software projects. In *Proc. of the European Conf. on Software Maintenance and Reengineering*, Szeged, Hungary, pp. 441–446. IEEE Computer Society.

J. Nonnen; D. Speicher; and P. Imhoff (2011) Locating the meaning of terms in source code research on "term introduction". In *Proc. of the 18th Working Conf. on Reverse Engineering*, Limerick, Ireland, pp. 99–108. IEEE Computer Society.

D. Posnett; A. Hindle; and P. Devanbu (2011) A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, Honolulu, Hawaii, pp. 73–82. ACM.

R Development Core Team (2008) *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

D. Raţiu (2009) *Intentional Meaning of Programs.* Ph.D. thesis, Technische Universität München.

V. Rajlich and N. Wilde (2002) The role of concepts in program comprehension. In *Proc. 10th Int'l Workshop on Program Comprehension*, Paris, France, pp. 271–278. IEEE.

P. A. Relf (2004) Achieving software quality through identifier names. In *Proc. of Qualcon 2004*, South Australia.

P. A. Relf (2005) Tool assisted identifier naming for improved software readability: an empirical study. In *Int'l Symp. on Empirical Software Engineering*, Noosa Heads, Queensland, Australia, pp. 53–62. IEEE.

B. Santorini (1990) Part-of-speech tagging guidelines for the Penn Treebank Project. Technical Report MS-CIS-90-47, Department of Computer and Information Science, University of Pennsylvania. URL `http://repository.upenn.edu/cis_reports/570/`.

J. Singer and C. Kirkham (2008) Exploiting the correspondence between micro patterns and class names. In *Int'l Working Conf. on Source Code Analysis and Manipulation*, Beijng, China, pp. 67–76. IEEE.

Sun Microsystems (1999) Code conventions for the Java programming language. `http://java.sun.com/docs/codeconv`.

G. Suryanarayana; G. Samarthyam; and T. Sharma (2014) *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann.

A. A. Takang; P. A. Grubb; and R. D. Macredie (1996) The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167.

A. Taylor; M. Marcus; and B. Santorini (2003) The Penn Treebank: An overview. In A. Abeillé (ed.), *Treebanks: The state of the art in syntactically annotated corpora*, chapter 1, pp. 5–22. Kluwer.

The Eclipse Foundation (2007) Development conventions and guidelines. `http://wiki.eclipse.org/Development_Conventions_and_Guidelines`13).

K. Toutanova; D. Klein; C. Manning; and Y. Singer (2003) Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of HLT-NAACL*, Edmonton, Canada, pp. 252–259.

A. Vermeulen; S. W. Ambler; G. Bumgardner; E. Metz; T. Misfeldt; J. Shur; and P. Thompson (2000) *The Elements of Java Style*. Cambridge University Press.

K. D. Welker; P. W. Oman; and G. G. Atkinson (1997) Development and application of an automated source code maintainability index. *Journal of Software Maintenance*, 9(3):127–159.

D. A. Wheeler (2008) SLOCCount. `http://www.dwheeler.com/sloccount/`.

Appendices

Appendix A

Glossary

Aggressive Tokeniser An aggressive tokeniser seeks to split identifier names into the tokens the developer included in names and uses techniques to identify elided and implied typographical boundaries, boundaries in single case names (such as `ALTORENDSTATE`) and to identify tokens containing digits.

Cipher Ciphers are names composed of single letter abbreviations, commonly of a specific type such as `int i`, that are used to represent generic entities.

Code Smell A code smell is a feature of source code that indicates an underlying design weakness. Code smells both motivate refactoring and identify the steps required to refactor the source code.

Conservative Tokeniser A conservative tokeniser is one that splits names into tokens using the unambiguous typographical boundaries of the lower case to upper case transition and separator characters, e.g. `classnamePrefix` is tokenised to `classname`, `prefix`.

Content Type The lexical content of individual tokens in a name, e.g. whether tokens are abbreviations, specialised abbreviations familiar to software practitioners, acronyms or words.

Design Smell A design smell is a poor or unconventional solution to a common design problem that compromises source code quality and maintainability.

EJS "The Elements of Java Style" Vermeulen *et al.* (2000). A handbook of Java programming conventions, including naming conventions, drawn from the practice of developers

at Rogue Wave Software.

Hard Word A term coined by Feild *et al.* (2006). The tokens of a name that are marked by the typographical word boundaries of the lower to upper case transition and separator characters, e.g. the name `HTMLEditorKit` consists of the hard words `HTMLEditor` and `Kit`. Hard words result from the use of a conservative tokeniser and may consist of one or more *soft words*.

JLS "The Java Language Specification" Gosling *et al.* (2014). The specification of the Java programming language that includes naming conventions.

Multi-Part Name A name consisting of two or more constituent parts where at least one is non-phrasal, e.g. `ER_CANT_CREATE_URL` where 'ER' is most likely an abbreviation of 'error' used as a prefix to identify error messages.

Multiple Phrase Name A name that consists of two or more, sometimes disjoint, natural language phrases, e.g. `ThreadReferenceImpl_`
`Unable_to_pop_the_requested_stack_frame_from_the_call_stack__`
`Reasons_include__`
`The_requested_frame_was_the_last_frame_on_the_call_stack__`
`The_requested_frame_was_the_last_frame_above_a_native_frame__`
`12`

Neologism A neologism is a new word or phrase. Neologisms may be entirely novel, a novel use of an existing word, such as the use of *text* as a verb when mobile phones were introduced, or derived from existing words. Common mechanisms used to derive neologisms include the application of an affix not usually used with the donor word, e.g. *precisify*, and word blending, e.g. *brunch*.

Non-Phrasal Name Non-phrasal names are composed of words, but in an order or combination that is apparently nonsensical e.g. `isShowLines` and `manual_lazy_haves`

Oversplit Oversplitting is used to describe two distinct phenomena. (1) Software used to split identifier names may split the name into more words than optimal. (2) Developers may insert typographical boundaries between components of words that are then ac-

cepted by naive splitting techniques as separate words. For example, some developers introduce a typographical boundary following the prefix 'sub' as in `subMenu`.

Redundant Prefix A form of Hungarian Notation sometimes used in Java names where a single letter is used to indicate either the role of an identifier name, or its type. For example a field name might be prefixed with 'f' or a boolean variable with 'b'.

Refactoring Refactoring is the process of revising source code to improve its quality and maintainability while preserving external behaviour.

Separator Character A non-alphabetical character used to separate tokens in a name, e.g. the underscore in `ER_CANT_CREATE_URL`.

Sigil A sigil is a symbol used as a prefix to a name in some programming languages to differentiate between names and bare words, or between the role of an identifier name. For example, in Perl the identifier declarations `$a`, `@b` and `%c` are respectively a scalar, an array and a hash.

Species A species is a category of identifier name within a programming language. In Java the species include class, interface, constructor, method, field, formal argument and local variable.

Soft Word A term coined by Feild *et al.* (2006). The tokens of a name that are single dictionary words, abbreviations or acronyms that may or may not be divided from another soft word by a typographical boundary, e.g. `HTMLEditorKit` consists of the soft words `HTML`, `Editor` and `Kit`. Where a name follows typographical conventions, hard words and soft words are identical. Where typographical conventions are not followed, soft words are identified by an aggressive tokeniser.

Type Acronym A type acronym is a variable name that is an acronym of its declared type, e.g. `StringBuilder sb`.

Upper case to lower case (UCLC) boundary An implied token boundary that occurs in a name where there is a transition from two or more upper case characters to lower case. The boundary is ambiguous, e.g. `HTMLEditorKit` and `PBinitialize`, and a problem that aggressive tokenisers need to solve.

Appendix B

Corpus

Table B.1 lists the 60 projects in the `INVocD` corpus. For each project the version and size in thousands of source lines of code (KSLOC)[1] are given, and some characteristics of the project. The corpus contains some 16.5 MSLOC of code, 5 million identifier name declarations, 800,000 unique identifier names, and 25,000 unique tokens.

Table B.1: Corpus of 60 FLOSS Java Projects

Project	Version	KSLOC	Desktop Application	Project Management	Language Tool	Programmer Tool	Programming Language	IDE	SDK	Library	Server
AirTODO	1.27 final	13	*	*							
ANT	1.7.1	117				*					
ANTLR	3.2	43			*					*	
ArgoUML	0.30.2	203	*								
ASM	3.3	23			*					*	
AspectJ	1.6.9	382				*					
Azureus (Vuze)	4.5.02	491	*								
BCEL	5.2	24			*					*	
Beanshell	2.0b4	26					*				
BlueJ	3.0.2	76	*					*			
BORG Calendar	1.7.3	25	*	*							
Cactus	1.8.1	22				*					
Cobertura	1.9.4.1	51				*					

[1] Generated using David A. Wheeler's 'SLOCCount' (Wheeler, 2008)

Project	Version	KSLOC	Desktop Application	Project Management	Language Tool	Programmer Tool	Programming Language	IDE	SDK	Library	Server
Derby	10.6.1.0	619								*	*
Eclipse	3.6.0	2,296	*					*	*		
E-Gantt	0.5.3a	11		*						*	
FindBugs	1.3.9	109	*			*					
FreeMind	0.9.0RC9	52	*								
GanttProject	2.0.10	51	*	*							
Geronimo	3.0-M1	190									*
Google Web Toolkit	2.0.4	391							*		
Greenfoot	1.5.6	108	*								
Groovy	1.7.4	120					*				
Hibernate	3.5.5-final	328								*	
JabRef	2.6	98	*								
JasperReports	3.7.4	194								*	
Java 6 library	6u20-b02	890								*	
JavaCC	5.0	17			*						
JBoss	6.0.0 M4	281									*
JDK	6u21 fcs	2,327							*		
jEdit	4.3.2	110	*					*			
Jetty	7.2.0	16									*
JFreeChart	1.0.13	95								*	
Jin	2.1.4.1	34	*								
JRuby	1.5.2	169					*				
JUnit 4	4.8.2	6				*					
Jython	2.5.2 beta 1 (svn r7109)	210					*				
Kawa	1.10	114					*				
Log4J	1.2.16	25								*	
Lucene	3.0.2	132								*	
Maven	2.2.1	38				*					
Memoranda	1.0-rc3.1	22	*	*							
MindRaider	7.6	41	*								
MPXJ	4.0.0	73		*						*	
MultiJava	1.3.2	89			*						
NetBeans	6.9.1	4,217	*					*			
OpenProj	1.4	79	*	*							
Polyglot	1.3.5	47			*						
Rapla	1.3.2	59	*	*							
Rhino	1.7R2	62					*				
Spring	3.0.3	318								*	
Stripes	1.5.3	18								*	
Struts	2.2.1	144								*	

Project	Version	KSLOC	Desktop Application	Project Management	Language Tool	Programmer Tool	Programming Language	IDE	SDK	Library	Server
Tapestry	5.1.0.5	97								*	
Tomcat	6.0.29	165									*
Velocity	1.6.4	37				*				*	
Wicket	1.4.10	141									*
Xalan-Java	2.7.1	176			*					*	
Xerces-j	2.10.0	126			*					*	
XOM	1.2.6	48			*					*	

Appendix C

Database Schema

This appendix details the database schema used in the `INVocD` database. All database tables belong to the schema SVM, and all table names must be prefixed with SVM.

C.1 Program Entities

The database schema centres on the program_entities table, which follows the AST using the container_uid and entity_uid columns to identify the current and containing program entities. The '`fk`' suffix to a column name identifies foreign keys. The 'is_anonymous' column identifies program entities without an identifier name. In those cases the identifier name key points to the identifier '`#anonymous#`'.

PROGRAM_ENTITIES
program_entity_key
project_key_fk
package_key_fk
identifier_name_key_fk
container_uid VARCHAR(255)
entity_uid VARCHAR(255)
species_name_key_fk
type_name_key_fk
method_signature_key_fk
is_anonymous BOOLEAN
file_name_key_fk INT
is_array BOOLEAN
is_loop_control_var BOOLEAN
start_line_number INT
start_column INT
end_line_number INT
end_column INT

C.2 Names and Tokens

Unique identifier names are stored with a count of their component words or tokens, and are referenced by both program entities and type names.

IDENTIFIER_NAMES
identifier_name VARCHAR(255)
identifier_name_key
components INT

Unique component words are recorded and linked to identifier names through a cross reference table that records the position within the identifier name where the component word is found.

COMPONENT_WORDS
component_word VARCHAR(255)
component_word_key

COMPONENT_WORDS_XREF
component_word_key_fk
identifier_name_key_fk
position INT

C.3 Type Names, Super Classes and Super Types

Unique type names are stored with a reference to the identifier name used to specify the type. The fully qualified name of the type is recorded if it can be resolved.

TYPE_NAMES
type_name VARCHAR(255)
type_name_key
identifier_name_key_fk

Type names involved in inheritance are cross referenced with program entities using two tables one of which corresponds to class based inheritance and the other to type based inheritance.

SUPER_CLASS_XREF
sub_class_entity_key_fk
super_class_name_key_fk

SUPER_TYPE_XREF
sub_type_entity_key_fk
super_type_name_key_fk

C.4 Method Signatures

Unique method signatures are recorded for constructor and method program entities.

METHOD_SIGNATURES
method_signature_key
method_signature VARCHAR(2048)

C.5 Species and Modifiers

A read only set of species is used to classify program entities. The values of the `species_name` column are: annotation member, annotation, class, constructor, enum constant, enum, field, formal argument, initialiser, interface, label name, local, local class, member class, method, and nested interface.

SPECIES
species_name_key
species_name VARCHAR(20)

A read only set of modifiers is provided and multiple modifiers linked to program entities through a cross reference table. The members are: `abstract`, `final`, `native`, `private`, `protected`, `public`, `static`, `strictfp`, `synchronized`, `transient` and `volatile`.

MODIFIERS
modifier VARCHAR(20)
modifier_key

MODIFIERS_XREF
modifier_key_fk
program_entity_key_fk

C.6 Projects, Packages and Files

Unique package names are stored in a table, and cross referenced with projects. This is done to save space when storing multiple projects, which are distinguished by a project name and

version pair

PACKAGE_NAMES
package_name_key
package_name VARCHAR(255)

PACKAGES
package_name_key_fk
package_key
project_key_fk

PROJECTS
project_key
project_name VARCHAR(255)
project_version VARCHAR(255)

Unique file names are recorded for the program entities where they are found.

FILES
file_name VARCHAR(255)
file_name_key

Any individual file is identified by the combination of project name and version, package name and file name.

Appendix D

Software and Data Created During the Research

This appendix provides details of the software and data created during the course of the research reported in this dissertation that is available for download. Details of the licence for each binary package, and `INVocD` are provided with the respective downloads. All the open source versions of software are licensed using version 2.0 of the Apache Licence, with the exception of `MDSC`, which uses the GPL v3 licence with 'classpath exception'.

D.1 Software

D.1.1 `INTT`

The version of `INTT` (v0.2.0) described in Chapter 4 is available in binary form from `http://oro.open.ac.uk/28352/` Included in the zip file with `INTT` are the 7 sets of 4,000 manually tokenised names used to test `INTT` and 1.4 million records of the unique identifier names extracted from the corpus.

`INTT` is still being developed and recent versions are open source software. The repository is at `https://github.com/sjbutler/intt`

D.1.2 `JIM`

JIM is a tool for mining identifier names and metadata from Java source code. `JIM` is described in Chapter 3 and was used to create the `INVocD` corpus.

Three versions of JIM are available in binary form:

- v0.4.2 with parsers for Java v6 and below. Compiled for Java v6.
- v0.5.9 with parsers for Java v7 and below. Compiled for Java v7.
- v0.5.9 with parsers for Java v7 and below. Compiled for Java v8.

Compressed archives containing the JAR files can be downloaded from http://www.facetus.org.uk/research/jim

JIM is still being developed, and recent versions are open source software. The repository can be found at https://github.com/sjbutler/jim

D.1.3 JIMdb

JIMdb is a component of JIM that mediates access to the database, and has an API for extracting names and metadata. JIMdb is made available as a separate library to provide access for analytic tools to databases created by JIM. The repository is at https://github.com/sjbutler/jimdb

D.1.4 MDSC

MDSC is a spell checking library designed to use multiple dictionaries. MDSC is described in Chapter 6. MDSC is open source software and the repository is at https://github.com/sjbutler/mdsc

D.1.5 Nominal

Nominal, the library for specifying and evaluating adherence to naming conventions described in Chapter 7, is open source software. The repository is at https://github.com/sjbutler/nominal

D.2 Data

D.2.1 INVocD

INVocD is available from http://oro.open.ac.uk/36992/ as a download that includes Apache Derby and SQLlite versions of the database, and a SQL dump of the database, with DDL

files, that can be imported into other SQL database systems.

Appendix E

Penn Treebank Tags

The following lists of Penn Treebank tags are adapted from those given in Santorini (1990) and Taylor *et al.* (2003).

E.1 Part of Speech Tags

CC - Coordinating conjunction

CD - Cardinal number

DT - Determiner

EX - Existential 'there'

FW - Foreign word

IN - Preposition or subordinating conjunction

JJ - Adjective

JJR - Adjective, comparative

JJS - Adjective, superlative

LS - List item marker

MD - Modal

NN - Noun, singular or mass

NNS - Noun, plural

NNP - Proper noun, singular

NNPS - Proper noun, plural

PDT - Predeterminer

POS - Possessive ending

PRP - Personal pronoun

PRP$ - Possessive pronoun

RB - Adverb

RBR - Adverb, comparative

RBS - Adverb, superlative

RP - Particle

SYM - Symbol

TO - to

UH - Interjection

VB - Verb, base form

VBD - Verb, past tense

VBG - Verb, gerund or present participle

VBN - Verb, past participle

VBP - Verb, non-3rd person singular present

VBZ - Verb, 3rd person singular present

WDT - Wh-determiner

WP - Wh-pronoun

WP$ - Possessive wh-pronoun

WRB - Wh-adverb

E.2 Phrase/Chunk Tags

ADJP - Adjective Phrase.

ADVP - Adverb Phrase.

CONJP - Conjunction Phrase.

FRAG - Fragment.

INTJ - Interjection. Corresponds approximately to the part-of-speech tag UH.

NP - Noun Phrase.

PP - Prepositional Phrase.

VP - Verb Phrase.

WHADJP - Wh-adjective Phrase. Adjectival phrase containing a wh-adverb, as in how hot.

WHAVP - Wh-adverb Phrase. Introduces a clause with an NP gap. May be null (containing the 0 complementizer) or lexical, containing a wh-adverb such as how or why.

WHNP - Wh-noun Phrase. Introduces a clause with an NP gap. May be null (containing the 0 complementizer) or lexical, containing some wh-word, e.g. who, which book, whose daughter, none of which, or how many leopards.

WHPP - Wh-prepositional Phrase. Prepositional phrase containing a wh-noun phrase (such as of which or by whose authority) that either introduces a PP gap or is contained by a WHNP.